Automatic Verification of Applet Interaction Properties

NIKLAS GAWELL

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2009
Automatic Verification of Applet Interaction Properties

NIKLAS GAWELL

Master’s Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2009
Supervisor at CSC was Dilian Gurov
Examiner was Johan Håstad

TRITA-CSC-E 2009:128
ISRN-KTH/CSC/E--09/128--SE
ISSN-1653-5715
Abstract

This masters thesis concerns the evaluation of an algorithm for model-checking context-free processes against \( \mu \)-calculus expressions. The problem investigated in this thesis originates from a research project at the Royal Institute of Technology, Swedish Institute of Computer Science and INRIA, France, where a method for control flow analysis was developed.

During the work with this thesis, a small alteration of the method was discovered, which gives much better performance of the algorithm. The method, the alterations and the algorithm to solve the model checking problem, as well as an implementation is described in this thesis.

Sammanfattning

Detta examensarbete i datalogi avhandlar en utvärdering av en algoritm för model-checking av kontextfria processer under \( \mu \)-kalkyl.

Problemet har sitt ursprung i ett forskningsprojekt på Kungliga Tekniska Högskolan, Swedish Institute of Computer Science och INRIA, Frankrike. Forskningsprojektet hade tagit fram en metod för flödeskontrollanalys, men saknade en implementation av ett av stegen i metoden.

Under arbetet med denna uppsats har en mindre förbättring av metoden hittats som ger algoritmen i undersökningen mycket bättre prestanda. Metoden för flödeskontroll, modifieringen av den och algotritmen samt dess implementation beskrivs i denna uppsats.
Acknowledgments

I would like to thank my instructors, at CSC Dillian Gurov, Karl Meinke and Johan Hästad. Great thanks to Andres Matrinelli for help on CUDD.

I would also like to thank my fellow students who have supported me during this work, in particular Mikael Lagerkvist, who have provided much insights, help and suggestions.
# Contents

1 Introduction .............................................................................. 1

2 Preliminaries  ........................................................................... 3
  2.1 Model checking ................................................................. 3
  2.2 An abstract model ............................................................ 3
  2.3 Sets and automata .............................................................. 4
    2.3.1 Finite state automaton ............................................... 4
    2.3.2 Context-free processes ............................................. 4
  2.4 Lattice theory ................................................................. 6
    2.4.1 Partial orders ............................................................ 7
    2.4.2 Lattices ................................................................. 8
    2.4.3 Functions, monotonicity and fixed points ....................... 8
  2.5 Property specification languages ....................................... 9
    2.5.1 The propositional $\mu$-calculus .................................... 9

3 Verification of interaction properties .......................................... 14
  3.1 Modeling System Behavior .............................................. 14
  3.2 Program Structure .......................................................... 15
  3.3 Program Behavior ........................................................... 15
  3.4 Specifying properties ....................................................... 16
  3.5 Why not use existing tools? ............................................. 18

4 An algorithmic solution .......................................................... 19
  4.1 Algorithm outline ............................................................ 19
  4.2 Property transformers ..................................................... 20
    4.2.1 Computing the fixed points ....................................... 22
  4.3 Evaluating resulting property transformers .......................... 23
  4.4 A detailed example .......................................................... 23
  4.5 Correctness and complexity ............................................. 25
  4.6 Observations ................................................................. 25

5 Implementation ........................................................................ 27
Chapter 1

Introduction

In any line of activity that deals with production it is important to verify that the products are of good quality and that they meet the demands placed on them. This is certainly the case in software development as well. During the years several different methods have been suggested and used. Deductive reasoning, simulation and testing are a few. They all have advantages and disadvantages. Deductive reasoning is exact, but it is hard to analyze larger systems, such as a network protocol. Simulation and testing can cope with such systems, but for most cases the number of possibilities to test is too great for the methods to provide good confidence in the quality. One method which tries to resolve this problem is model checking.

We will in this thesis explain how to use model checking to verify security properties of applet interaction. We will show how to build an abstract model of control flow and how to specify certain important properties using temporal logics. We will also present an algorithm that verifies the properties and analyze the performance of that algorithm. It will be evident that some adjustments of the method is required, and we will give a few suggestions on such adjustments.

The problem we are trying to solve through this thesis originates from a research project shared by Swedish Institute of Computer Science, The Royal Institute of Technology Stockholm and INRIA Sophia Antipolis in France. The authors studied security verification of JavaCard applets, specially control flow behavior in order to guarantee absence of information leaks, their methods and results are presented in [4], [5], [6] and [8]. The authors developed a method and a tool kit for analyzing JavaCard programs but it turned out that the size of the models and properties to verify were too large for the existing tools to handle. Without being able to verify the properties of the models, the method is interesting but not very useful. During the work with this thesis, we implemented an algorithm that verifies the property, and we investigated
if the algorithm was efficient enough to be used in real world examples.

The algorithm was first described in a [1] by Burkhart and Steffen, and was later extended by the same authors in [2]. We implemented the algorithm using C++ and ran some tests on it. The algorithm proved to be much better than the tools previously used, but still not quite adequate to solve a full scale problem in the original formulation. We found a few modifications of the method of formulating the problem, and with those modifications the algorithm solved the problem with reasonable amount of resources.

We present the method for formulating the problem in chapters 2 and 3, the algorithm in chapter 4, the implementation of the algorithm in chapter 5 and some measurements on the performance of the implementation in this thesis in chapter 6.
Chapter 2

Preliminaries

In this chapter we introduce the concepts that we need in order to solve the problems we are faced with. We present how the model is represented and how we specify the properties to be investigated.

2.1 Model checking

Model checking is a procedure of verifying that for instance a computer program is working as intended. Unlike popular methods like unit tests model checking is a method of verification. Assuming you do it properly you know that what you have constructed a program that works.

Model checking is a three step process.

Modeling First we create a model of the system we wish to verify. This may be done in different ways. Sometimes it is just a question of translation or compilation. Sometimes we have to abstract parts of our system and make a model of that abstraction.

Specification We need to specify the properties our system must satisfy. Often some logic formalism is used, in software systems it is common to use a temporal logic.

Verification The verification is the final step, where we check if the given model has the specified properties.

2.2 An abstract model

We consider a model to be an abstraction of a system we wish to investigate. Depending on what type of properties we wish to prove about our system
we may adopt different modeling techniques. Since model checking often is concerned with proving properties of an automaton, it’s common to create a model of the system based upon some transition system scheme.

2.3 Sets and automata

Set theory will be used all through this thesis, so we start of by giving a brief summary of what terms and notation we will use. A set is a collection of unique elements and is in this thesis denoted by a capital letter.

We denote the set that has no members ∅.

The quantifiers we will use are for all elements of a, ∀ and, for at least one element, ∃.

We write $A \subseteq B$ when $A$ is a subset of or equal to $B$, and $A \subset B$ when $A$ is a proper subset of $B$.

A special set, that will be of interest to us is the power set. The power set of $A$ is the set of all subsets of $A$ and is denoted $2^A$.

Example 2.1. The powerset of $\{a, b, c\}$ is $2^{\{a, b, c\}} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

A notation that will be frequently used is $V^*$, which denotes the set of all strings, including the empty string $\epsilon$ over the alphabet $V$.

Example 2.2. $\{a,b,c\}^* = \{a,ab,abc,ac,aa,babc,aac,ab,ba,bc,bb,bbabc,bc,...\}$.

2.3.1 Finite state automaton

A common structure used in computer science is the finite state automaton, FSA.

Definition 2.3 Finite state automaton. A finite state automaton is a tuple

$P \triangleq (V, I, \rightarrow)$.

Where $V$ is a set of states, $I$ is a set of input symbols and $\rightarrow$ is a set of transitions

$\rightarrow \subseteq (V \times I^+) \times (V)$

$I^+$ is the set of input symbols and the empty string, $\epsilon$

$I^+ \triangleq \{\epsilon\} \cup I$.

2.3.2 Context-free processes

A context-free process is similar to a push down automaton, PDA. The difference is that a PDA contains both a stack and a set of states, where a context-free process only contains a stack.
Push down automaton

A push down automaton is a well studied structure. It can be seen as an extension of the finite state automaton. The extension is the addition of a stack to the automaton. Instead of just reading a symbol of the input when changing states the automaton looks both at the top of its stack, removing what might be there, and at the input symbol when determining what state to enter, and as it makes the transition it may put one or more new symbols on the stack.

**Definition 2.4 Push down automaton.** A push down automaton is a tuple

\[ P \triangleq (V, S, I, \Sigma, \rightarrow) \]

Where \( V \) is a set of states, \( S \) is a stack, \( I \) is a set of input symbols, \( \Sigma \) is a set of stack symbols and \( \rightarrow \) is a set of transitions

\[ \rightarrow \subseteq (V \times \Sigma^+ \times I^+) \times (V \times \Sigma^*) \]

The set \( \Sigma^+ \) is the set of stack symbols and the bottom symbol \# 

\[ \Sigma^+ \triangleq \{\#\} \cup \Sigma \]

The set \( I^+ \) is the set of input symbols and the empty string, \( \epsilon \)

\[ I^+ \triangleq \{\epsilon\} \cup I \]

The transitions are written as: \( \langle v, \sigma, i \rangle \rightarrow \langle v', \sigma^* \rangle \) where \( v \) is the state of the automaton, \( \sigma \) is the symbol on top of the stack, \( i \) is the current input symbol. \( v' \) is the next state of the automaton, and \( \sigma^* \) is the string of stack symbols to put on the stack.

**Rewrite systems**

In order to define a context-free process we need to make a few definitions. The following definitions are taken from [2].

**Definition 2.5 Labeled transition graph.** A labeled transition graph is a triple

\[ T \triangleq (S, Act, \rightarrow) \]

Where \( S \) is a set of states, \( Act \) is a set of labels, and \( \rightarrow \subseteq S \times Act \times S \) is the set of transition relations.

This is fairly similar to standard finite state automaton. But while finite state automaton have a finite set of states, a labeled transition graph may be infinite. Since we are interested in special kinds of infinite transition systems, we can be more precise than the definition above.
**Definition 2.6** Labeled rewrite system. A labeled rewrite system is a triple 
\[ \mathcal{R} \triangleq (V, \text{Act}, R). \]

Where \( V \) is an alphabet, \( \text{Act} \) is a set of labels, and \( R \subseteq V^* \times \text{Act} \times V^* \) is a finite set of rewrite rules. If \( \mathcal{R} \) is of the form \( R \subseteq V \times \text{Act} \times V^* \) the system is said to be alphabetic.

We will write \( u \xrightarrow{a} v \) as an alternative to \((u, a, v) \in R\).

**Definition 2.7** Prefix rewrite transition. Let \( \mathcal{R} = (V, \text{Act}, R) \) be a rewrite system, then a prefix relation on that system is defined by 
\[ \xrightarrow{\mathcal{R}} \triangleq \{(uw, a, vw) | (u \xrightarrow{a} v) \in R, w \in V^*\}. \]

A labeled transition graph \( T = (V^* \times \text{Act}, \xrightarrow{\mathcal{R}}) \) is called a prefix transition graph of \( \mathcal{R} \), if the transition relations \( \xrightarrow{\mathcal{R}} \) are prefix relations.

**Definition 2.8** Context-free process. An alphabetic prefix rewrite system is called a context-free system. A context-free process is a rooted prefix transition graph of a context-free system.

We will later show that a context-free process will suffice to model the kind of control flow behavior we are interested in.

The reason why context-free processes, PDAs and such are interesting is that they provide a finite description of possibly infinite structures. Still they are fairly simple models.

One important issue we haven’t addressed yet is how we go about building these context-free processes. The reason for this is that the strategy for doing this depends on the kind of property we want to verify. So before we know how to build our context-free process we need to determine what we are interested in. In chapter 3 we explain how to build context-free processes from our model of applets.

### 2.4 Lattice theory

Before we move on to property specification languages we will need some basic lattice theory to understand some of the concepts of the specification language. Let us start with some definitions.
2.4.1 Partial orders

Definition 2.9 Partial order relation. A partial order relation \( \sqsubseteq \) over a set \( A \) is a binary relation that has the properties:

\[
\begin{align*}
a \sqsubseteq a, \quad & \text{(2.1)} \\
a \sqsubseteq b \land b \sqsubseteq a \rightarrow a = b, \quad & \text{(2.2)} \\
\text{and} \quad & \text{(2.3)} \\
a \sqsubseteq b \land b \sqsubseteq c \rightarrow a \sqsubseteq c.
\end{align*}
\]

Definition 2.10 Partially ordered set. A set \( A \) with a partial order relation is called a partially ordered set or a poset. When the ordering relation is not clear from the context \((A, \sqsubseteq)\) denotes the poset, but often only \( A \) is used.

Definition 2.11 Least upper bound and greatest lower bound. Greatest lower bound \( \bot \) of \( A \) is the greatest element that satisfies \( \forall a. \bot \sqsubseteq a \) Least upper bound \( \top \) of \( A \) is the least element that satisfies \( \forall a. a \sqsubseteq \top \)

Not every poset has \( \bot \) or \( \top \). If \( \bot \) or \( \top \) do exist, they do not need to belong to \( A \), but they will be unique.

A sequence of elements \( a_1, a_2, a_3, \ldots \in A \) fulfilling \( a_1 \sqsubseteq a_2 \sqsubseteq a_3 \ldots \) is called an increasing chain. If \( A \) has a least element \( \bot \in A \) \( (\forall a \in A. \bot \sqsubseteq a) \) and every increasing chain with elements in \( A \) has an upper bound \( \top \), the poset \( A \) is a complete partial order, CPO.

Example 2.12. The rational numbers \( \mathbb{Q} \) form a poset under the ordering \( \leq \).

Example 2.13. The set \( \{1, 2, 3\} \) form a complete partially ordered set under the ordering relation \( \leq \) since \( \bot = 1 \) and \( 1 \leq 2, 2 \leq 3, 1 \leq 2 \leq 3 \) and \( 1 \leq 3 \) all have a least upper bound \( 3 \).

We also define two operations on partially ordered sets.

Definition 2.14 Binary operation meet. The binary operation meet on elements \( a \) and \( b \) of a partially ordered set \( A \) will give the greatest lower bound if such an element exists. We will denote the meet operation by \( \sqcap \)

\[
a \sqcap b = c \quad \text{where} \quad a \sqsubseteq c \land b \sqsubseteq c \land \forall d.(a \sqsubseteq d \land b \sqsubseteq d \rightarrow c \sqsubseteq d).
\]

Definition 2.15 Binary operation join. The binary operation join on elements \( a \) and \( b \) of a partially ordered set \( A \) will give the least upper bound if such an element exists. We will denote the join operation by \( \sqcup \)

\[
a \sqcup b = c \quad \text{where} \quad c \sqsubseteq a \land c \sqsubseteq b \land \forall d.(d \sqsubseteq a \land d \sqsubseteq b \rightarrow d \sqsubseteq c).
\]

We will use the operations on subsets of posets aswell

\[
\sqcup \{a, b, \ldots, z\} = a \sqcup b \sqcup \ldots \sqcup z.
\]
2.4.2 Lattices

In the analysis of the coming algorithm we will need to know a little about an algorithmic structure called a lattice.

**Definition 2.16 Lattice.** A lattice is a partially ordered set where $\sqcup$ and $\sqcap$ are defined and exists.

A lattice is called complete if all increasing chains have a least upper bound. In particular a lattice on a finite set is complete.

2.4.3 Functions, monotonicity and fixed points

We will be interested in functions over lattices.

**Definition 2.17 Monotone function.** A function $f : (A, \sqsubseteq_A) \to (B, \sqsubseteq_B)$ is called monotone if

$$a \sqsubseteq_A b \to f(a) \sqsubseteq_B f(b).$$

**Definition 2.18 Continuous function.** A function $f : (A, \sqsubseteq_A) \to (B, \sqsubseteq_B)$ is called continuous if for every increasing chain $C$ of $A$

$$f(\sqcup C) = \sqcup f(C).$$

Since we are only interested in finite posets it will be of some importance for us to know that monotonicity and continuity coincide for finite posets.

**Definition 2.19 Fixed point.** A function $f : A \to A$ has a fixed point at $a \in A$ if $f(a) = a$. The least $a \in A$ with this property is called the least fixed point of $f$ and is denoted $\mu f$. The greatest such element $b \in A$ is called the greatest fixed point, and is denoted $\nu f$,

$$\nu f = a \iff f(a) = a \land \forall x \in A. (f(x) = x \to x \sqsubseteq a)$$

and

$$\mu f = a \iff f(a) = a \land \forall x \in A. (f(x) = x \to x \sqsubseteq a).$$

Now we are ready to present the following important results.

**Theorem 2.20 Knaster-Tarski fix point theorem.** Let $L$ be a complete lattice and $f$ be a monotone function on $L$. Then

$$\mu f = \sqcap \{x \in L | f(x) \sqsubseteq x\}$$

and

$$\nu f = \sqcup \{x \in L | x \sqsubseteq f(x)\}.$$
Theorem 2.21. Let $L$ be a complete lattice and $f : L \rightarrow L$ be a continuous function. Then

$$\mu f = \bigcup_{n \geq 0} f^n(\bot)$$

and

$$\nu f = \bigcap_{n \geq 0} f^n(\top).$$

These two theorems tell us when to expect to find fixed points, and how to compute them. If we know that we have a continuous function $f$ on a complete lattice $L$ and are looking for the least fixed point. We simply start by taking the least element $\bot \in A$ and apply $f$ on it. If the result is different from $\bot$ we apply $f$ again until we end up with $f^{n+1}(\bot) = f^n(\bot)$, which is the fixed point we were looking for. If we were looking for the greatest fixed point, we start out with the greatest element $\top \in A$, $(\forall a \in A. a \sqsubseteq \top)$ and follow the same procedure.

2.5 Property specification languages

Again we have a choice between several different languages for specifying properties of our systems. The class of languages most often considered in model checking is temporal logics. Temporal logics deal with events in time, they let you express “when $a$ happens will $b$ always be true” for example.

2.5.1 The propositional $\mu$-calculus

The propositional $\mu$-calculus is a well studied variant of temporal logic. It combines ordinary logic connectives, like or, with temporal and fix point operators. There are several versions of the $\mu$-calculus, we will use the one of [2] combined with atomic propositions suggested in [5]. A $\mu$-calculus formula is interpreted in the context of a transition system, such as a context-free process. A formula evaluates to the states in the transition system where the formula holds. A $\mu$-calculus formula $\Phi$ is built as follows:

- $tt, ff, p, \neg p, X$ are formulas.
- If $\Phi_1$ and $\Phi_2$ are formulas, $\Phi_1 \land \Phi_2$ and $\Phi_1 \lor \Phi_2$ are formulas.
- If $\Phi$ is a formula and $a$ is a label in the transition system then $(a)\Phi$ and $[a]\Phi$ are formulas.
- If $X$ is a variable and $\Phi$ is a formula containing $X$, then $\mu X.\Phi$ and $\nu X.\Phi$ are formulas.
where \( tt \) and \( ff \) denote true and false respectively. True is an abbreviation for all states, and false stands for no states, \( p \) is an atomic proposition. An atomic proposition is a proposition about a state, for example “node \( n \) is an entry node” of a method call graph. \( X \) is a variable from the set \( Var \) of variables.

The semantics of a formula is defined recursively in the context of a transition system \( T = (V^*, Act, \rightarrow) \), a constant proposition function \( S : V^* \rightarrow 2^P \) where \( P \) is the set of all propositions of the language. \( S(\alpha) \) returns the propositions \( p \subseteq P \) that are true at \( \alpha \). In the case of the method call graphs, definition 3.1, \( S = \lambda_m \). We also need a function \( V : X \rightarrow 2^P \), called a valuation, which assigns to each variable \( X \in Var \), the set of states where the variable is true.
Definition 2.22. A state $\alpha \in V^*$ has the property $\Phi$, here denoted $\alpha \models_{V,S} \Phi$, when:

$\alpha \not\models_{V,S} tt$ ,

$\alpha \models_{V,S} p$ iff $p \in S(\alpha)$,

$\alpha \models_{V,S} \neg p$ iff $p \notin S(\alpha)$,

$\alpha \models_{V,S} X$ iff $\alpha \in V(X)$,

$\alpha \models_{V,S} \Phi_1 \lor \Phi_2$ iff $\alpha \models_{V,S} \Phi_1$ or $\alpha \models_{V,S} \Phi_2$,

$\alpha \models_{V,S} \Phi_1 \land \Phi_2$ iff $\alpha \models_{V,S} \Phi_1$ and $\alpha \models_{V,S} \Phi_2$,

$\alpha \models_{V,S} \langle a \rangle \Phi$ iff $\exists \alpha' \xrightarrow{a} \alpha'$ and $\alpha' \models_{V,S} \Phi$,

$\alpha \models_{V,S} [\langle a \rangle \Phi$ iff $\forall \alpha'. \alpha \xrightarrow{a} \alpha'$ implies $\alpha' \models_{V,S} \Phi$,

$\alpha \models_{V,S} \mu X. \Phi$ iff $\forall S \subseteq V^*. (\forall \beta \in V^*. \beta \models_{V\setminus S} \Phi$ implies $\beta \in S) \text{ implies } \alpha \in S$

or

$\alpha \models_{V,S} \nu X. \Phi$ iff $\exists S \subseteq V^*. (\forall \beta \in V^*. \beta \in S \text{ implies } \beta \models_{V\setminus S} \Phi$ and $\alpha \in S$.

The first few rules are intuitive. No state fulfills $ff$. Any state fulfills $tt$. A state fulfills variable $X$ if it is in the valuation of $X$. Any state that fulfills $\Phi_1$ or $\Phi_2$ fulfills $\Phi_1 \lor \Phi_2$. Analogously for the and operator. If there exists a transition labeled $a$ from state $\alpha$ which ends up in a state that fulfills $\Phi$ then $\alpha \in \langle a \rangle \Phi$. If it is true for all transitions labeled $a$ from state $\alpha$ that they end up in a state that fulfills $\Phi$ then $\alpha \in [\langle a \rangle \Phi$.

The $\mu$ and the $\nu$ are fix point operators. If $\sigma$ is any fix point operator then it is true that $\alpha \models \sigma X. \Phi$ iff $\alpha \models \Phi[\sigma X. \Phi / X]$, where $\Phi[\sigma X. \Phi / X]$ is the same formula as $\Phi$ but with $X$ replaced with $\sigma X. \Phi$. An “unfolding” of the formula, see example 2.23.

Example 2.23. The unfolding of simple fix point formula

$$\sigma X. p \land X \rightarrow \sigma X. p \land (\sigma X. p \land X).$$

The fix point $\mu$ is the least such point, and $\nu$ is the greatest. In order for us to say that an element is the least or the greatest we need an ordering. The order used in this context is set inclusion $\subseteq$. It is always true that $\mu X. \Phi \subseteq \sigma X. \Phi \subseteq \nu X. \Phi.$
According to the Knaster-Tarski fix point theorem every monotone function on a complete lattice has fixed points. Evaluation of formulas is monotone, if we restrict the use of negations. Which we have done, the only way to impose a negation is to negate an atomic proposition, and a negated atomic proposition is an atomic proposition. Using DeMorgan’s law and the dualities

\[\hat{\neg}\neg (a) \equiv [a] \neg \Phi,\]
\[\hat{\neg} (a) \equiv [a] \neg \Phi,\]
\[\hat{\neg} \sigma X. \Phi(X) \equiv \sigma X. \neg \Phi(\neg X).\]

any formula can be expressed using negations only on atomic propositions. If the formulas are finite, we can expect to find fixed points. Since the number of labels in our transitions systems are finite, will all formulas from here on be finite.

We will also impose a restriction. A variable in a formula must be bound by a fixed point operator. This restriction exists because the only way to alter a variable during the composition process, is if it is a variable in a fixed point. If it is not, it will never be updated, and thus not a variable, and we can replace it with a proposition.

**Definition 2.24 Bound variable.** A variable \(X\) is bound in \(\sigma X. \Phi(X)\) and is considered unbound otherwise. A formula containing only bound variables is called well named.

From now on all formulas will be well named.

A property of formulas that is of some interest is how many nested fixed points there are, or to be more specific, how many levels of maximal inside minimal inside greatest fixed points there are. To specify this notion we define the concept of alternation depth along the lines presented in [2].

**Definition 2.25 Alternation Depth.** Alternation depth is recursively defined as follows

- A formula containing no fixed point operators has alternation depth 0.
- A greatest fixed point formula has the alternation depth of the least fixed point sub formula with the highest alternation depth plus one.
- A least fixed point formula has the alternation depth of the great fixed point sub formula with the highest alternation depth plus one.

The following example will hopefully make it clear.
Example 2.26 Alternation Depth. Here are a few examples of formulas and their alternation depth:

\[
\begin{align*}
ad([a]X) &= 0 \\
ad(\nu X.[a]X) &= 1 \\
ad(\mu Y.\nu X.([a]X \land [b]Y)) &= 2 \\
ad(\nu Y.\nu X.([a]X \land [b]Y)) &= 1
\end{align*}
\]

The algorithm studied in this thesis uses the formulas in a somewhat different form than what has been presented up until now. It requires the formulas to be in simple form. When writing a formula in simple form the formula is broken down into a set of equations, where each equation is of the form

\[
\Phi_{Simple} = tt | ff | p | \neg p | X | X \lor X | X \land X | (a).X | [b].X | \mu Y.X | \nu Y.X (2.4)
\]

The equations \(\Phi_{Simple} = \mu Y.X\) and \(\Phi_{Simple} = \nu Y.X\) are often written \(\Phi_{Simple} \mu = X\) and \(\Phi_{Simple} \nu = X\). Constructing a system of equations from a formula is done by substituting each sub formula of an expression by a variable, and adding an equation to the system defining the variable as being equal to the sub expression it replaced. Repeat this until all equations are on the form 2.4. If we take one of the formulas of example 2.26 it will look like:

Example 2.27 Simple form. A formula written in simple form.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Simple form</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mu Y.\nu X.([a]X \land [b]Y))</td>
<td>(\Psi_1 \mu = \Psi_2)</td>
</tr>
<tr>
<td></td>
<td>(\Psi_2 \nu = \Psi_3)</td>
</tr>
<tr>
<td></td>
<td>(\Psi_3 = \Psi_4 \land \Psi_5)</td>
</tr>
<tr>
<td></td>
<td>(\Psi_4 = [b]\Psi_6)</td>
</tr>
<tr>
<td></td>
<td>(\Psi_5 = [a]\Psi_7)</td>
</tr>
<tr>
<td></td>
<td>(\Psi_6 = \Psi_1)</td>
</tr>
<tr>
<td></td>
<td>(\Psi_7 = \Psi_2)</td>
</tr>
</tbody>
</table>

The set of all \(\Psi_i\) associated with a formula \(\Phi\) in this way is called the closure of the formula, and will be denote \(CL(\Phi)\). The outermost variable, \(\Psi_1\) in this case, is called the root of the property. The number of sub formulas will be denote \(|\Phi|\). In the example \(|\Phi| = 7\).

For more information about \(\mu\)-calculus and model checking we recommend [3]. It is concerned with finite state systems and therefore you will not find anything about context-free processes, but it gives a nice introduction to the topic of model checking.
Chapter 3

Verification of interaction properties

This chapter gives a presentation of how the our problem is formulated and why there is a need for a model checker that solves it.

3.1 Modeling System Behavior

There are several different ways to model a system. In order to choose a model specification language we need to know what kind of properties we wish to prove. We are interested in interaction properties of applets. The interaction we model is calls from one component to another. In our case method calls. Since we are only interested in the order of calls, and where they call, we will start with a method call graph.

Definition 3.1 Method call graph, adopted from [4]. A method call graph \( m \in M \) is a tuple

\[
\begin{align*}
  m \triangleq (V_m, \rightarrow_m, \lambda_m, \mu_m).
\end{align*}
\]

Where \( M \) is the set of graphs of all methods of the system, \( V_m \) are the program points of \( m \), \( \rightarrow_m \subseteq V_m \times V_m \) are the transfer edges of \( m \) and \( \lambda_m : V_m \rightarrow T \) where \( T \) is the set of program point types, \( T \triangleq \{entry, seq, call, return\} \). \( \mu_m : V_m \rightarrow \wp(M) \) which maps every program point of type call to the method it calls.

A program point is a point in the program, perhaps a line, a block or a part of an expression. It becomes a point in our view when it has some interesting property in the control flow aspect. We will be interested in points where method calls are made, entry and return points of methods. In example 3.2 we show how to extract program points from a piece of code.

The program point types are interpreted as follows: The point entry is the entry point of a method, seq is a program point where no call or return takes
place. The program point call is a point where a method call takes place. The point return is a point where the execution of the method finishes.

The program points of the whole system is the set

\[ V \triangleq \bigcup_{m \in M} V_m. \]

Since we are interested in the behavior of the control flow, not the structure of the calls a method call graph does not suffice. We need something that models the, possibly infinite, behavior of a system. Push down automata are often used to model recursive languages. But we can use something a bit more simple. A context-free process.

### 3.2 Program Structure

Since we are concerned with verifying properties about control flow behavior of recursive applets we adopt the structure of a context-free process, defined in definition 2.8. We are interested in verifying properties like “a call to A will never be followed by the control eventually ending up in B”. A few things are worth noticing here. If we were only interested in the effects of a call to A, the method call structure would suffice. We would only need to make a simple tree search from A and see if we reached B. But a call to A might, when A returns the control flow, be followed by a call to B. This may also be solved using the control flow graph, with slight modifications, but if we phrase a question like “In any infinite sequence of calls there may only occur one call to A” we will find that the simple control flow graph will fail. We need a more powerful structure, the context-free process for example.

### 3.3 Program Behavior

Huisman et al showed in [5] and [4] that control flow behavior could be modeled as a context-free process. We will adopt a slightly different method, but for everything but return action, equation 3.3, it will be exactly the same.

\[ \begin{align*}
    v & \rightarrow v' \quad v \models \lnot \text{return} \\
    v & \leftarrow v' \\
    v_1 \xrightarrow{m_2} v'_1 \quad v_1 \models \lnot \text{return} & v_2 \models m_2 \land \text{entry} \\
    v_1 \xrightarrow{m_1 \text{ call } m_2} v_2 \cdot v'_1 \\
    v_1 \models \text{return} & v_1 \models m_1 \\
    v_1 \xrightarrow{m_1 \text{ ret}} \epsilon
\end{align*} \]

15
Each symbol $v \in V$ in the context-free process corresponds to a node in the graph representation. A state $\sigma \in V^*$ satisfies the atomic proposition $p$ if and only if $p = tt$ or $v \models p$. If $v_1$ is an entry node of method $m$ then the symbol $A$ corresponding to $v_1$, and all states beginning with $A$, satisfies the propositions $tt, m, entry$.

The rules 3.1, 3.1 and 3.2 are interpreted as follows:

- If there is a transition from $v$ to $v'$ and $v$ is not an entry node, we add the production $v \rightarrow v'$ to the context-free process, according to equation 3.1.

- If there is a $v_1 \xrightarrow{m_2} v_1'$ transition from $v_1$ to $v_1'$, $v_1$ is not a return node and $v_2$ is the entry node of $m_2$, equation 3.2 gives us $v_1 \xrightarrow{m_1 \ call \ m_2} v_2 \cdot v_1'$.

- If $v_1$ is a return node of $m_1$ then equation 3.3 gives $v_1 \xrightarrow{m_1 \ call \ m_2} v_2 \cdot v_1'$.

**Example 3.2.** A small Java program translated into control flow graph and a context-free process

<table>
<thead>
<tr>
<th>Java</th>
<th>Graph</th>
<th>CFP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>01 int m(int n) {</code></td>
<td>$V = {A, B}$</td>
<td>$A \xrightarrow{m \ call \ m} A \cdot B$</td>
</tr>
<tr>
<td><code>02 int a = 1;</code></td>
<td>$Act = {\epsilon, m \ call \ m}$</td>
<td>$A \xrightarrow{\epsilon} B$</td>
</tr>
<tr>
<td><code>03 if (n &gt; 1)</code></td>
<td>$\Rightarrow = {(A, m \ call \ m, A)}$</td>
<td>$\ xrightarrow{m ret} \epsilon$</td>
</tr>
<tr>
<td><code>04 a = m(n-1)*n;</code></td>
<td>$(A, \epsilon, B)$</td>
<td><code>05 return a;</code></td>
</tr>
<tr>
<td><code>06 }</code></td>
<td>$A \models m, entry$</td>
<td>$B \models m, return$</td>
</tr>
</tbody>
</table>

Program point $A$ is on line 02 and point $B$ on line 05. The actions for the method call graph are $\epsilon$ between $A$ and $B$ and $m \ call \ m$ from the method call on line 04. $A$ is an entry node and inside method $m$, and $B$ is a return node, and inside method $m$.

Using 3.2 we get $A \xrightarrow{m \ call \ m} A \cdot B$. Using 3.1 we get $A \xrightarrow{\epsilon} B$ and from 3.3 we get $B \xrightarrow{m \ ret} \epsilon$.

### 3.4 Specifying properties

To specify the safety properties to prove about the control flow Huisman et al chose a fragment of the $\mu$-calculus. Since they were interested in security
issues they did not need the \( \langle a \rangle \) operator and least fixed points. The calculus they used was given by:

\[
\Phi' = tt \mid ff \mid p \mid \neg p \mid X \mid \Phi \lor \Phi \mid \Phi \land \Phi \mid [b]\Phi \mid \nu X.\Phi.
\]

An important observation may be made here, the alternation depth of a formula on this form will always be 0 or 1, since there are no least fixed points.

In order to specify interesting properties Huisman et al defines a few patterns which help to express the kind of behaviors that are interesting,

\[
\text{Always } \Phi = \nu Z.\Phi \land [\text{Act}]Z. \tag{3.4}
\]

Where \( \text{Act} \) is the set of all labels in the context-free process. This is a fairly simple property. The intuitive meaning of it is \( \Phi \) is true and for all sequences of actions in \( \text{Act} \) it will continue to be true. The next one is also fairly simple,

\[
\text{Within } m \Phi = \neg m \lor (\text{Always } \Phi). \tag{3.5}
\]

The meaning of this pattern is, either the state does not satisfy \( m \), or it always satisfies \( \Phi \). So if a state is in \( m \) any state in any path from there will satisfy \( \Phi \),

\[
\text{CanNotCall } A \ B = \bigwedge_{m \in A} \bigwedge_{m' \in B} [m \text{ call } m'] ff. \tag{3.6}
\]

Where \( A \) and \( B \) are sets of methods. This property specifies that no method in \( A \) calls any method in \( B \).

Using these patterns a few interesting properties may be specified. We could, for instance, specify the property that within \( m \) the control flow will never end up in \( m' \);

\[
\text{Within } m (\text{CanNotCall } M \{ m' \}). \tag{3.7}
\]

Where \( M \) is the set of all methods in the system. This is a rather important property of a program, that no information is sent to \( m' \) as a result of a call to \( m \).

Given the partial Java program in example 3.2 the property \( \text{Within } A (\text{CanNotCall } M \{ B \}) \). The property express that no call to \( A \) will end up in \( B \),

\[
\neg A \lor \nu X.[A \text{ call } B] ff \land [B \text{ call } B] ff \land [\text{Act}] X. \tag{3.8}
\]

The program that does not contain calls from \( A \) to \( B \), a), will satisfy the property 3.8. But the other program, b), does not.
Example 3.3 Interaction properties of Java programs.

```java
a) int A(int n) {
    if (n = 0)
        return 1;
    return n*A(n-1);
}

int B(int n) {
    if (n < 2)
        return n;
    return B(n-2)+B(n-1);
}

b) int A(int n) {
    if (n < 2)
        return n;
    return A(n-1)+B(n-2);
}

int B(int n) {
    if (n < 2)
        return n;
    return A(n-1)+B(n-2);
}
```

Table 3.1. a) Simple Java example where no call to $A$ ends up in $B$. b) Simple Java example where a call to $A$ ends up in $B$.

### 3.5 Why not use existing tools?

This kind of problem is not new, and there exist a few tools for solving it. In [5] the authors used the model checker Alfred [7] which works on push down automata. Though Alfred solved the problem for small instances it failed to produce results on the full scale examples in [5].
Chapter 4

An algorithmic solution

Until 1992 it was unknown if $\mu$-calculus was decidable for context free processes. But Burkart and Steffen showed that a fragment of the calculus, alternation free modal $\mu$-calculus was decidable by giving an algorithm in [1]. In the same paper the authors hinted how to extend the algorithm to the full modal $\mu$-calculus. This was followed by an algorithm presented in 1995, also this time by Burkart and Steffen [2], giving an algorithm for the full calculus.

To solve the type of problems we are faced with the alternation free fragment is enough, but since the main problem in extending the algorithm to the full calculus is to prove the correctness of it, we will go for the full algorithm, since the proof already exists.

This chapter is devoted to explain the algorithm as cleanly as possible. We will try to explain and exemplify the important aspects and give an informal analysis of how and why it goes about solving the problem.

4.1 Algorithm outline

The algorithm takes a context-free process $C = (V, Act, \rightarrow)$ and a formula $\Phi$ written in $\mu$-calculus and calculates for what parts of the process the formula holds. The key concept in this algorithm is to calculate a set of functions called property transformers. A property transformer $[A]^{\Psi} : 2^{CL(\Phi)} \rightarrow \mathbb{B}$ is a function from post- to precondition that reflects the execution of a stack symbol $A \in V$. It is defined as

$$[A]^{\Psi}(\Delta) \triangleq \begin{cases} 1 & \text{if } \Psi \text{ is in the precondition and } \Delta \text{ is the post condition} \\ 0 & \text{otherwise.} \end{cases} \quad (4.1)$$

A nice feature with defining the property transformers in this way is that the transformers are elements of the boolean lattice consisting of all functions
from $2^{CL(\Phi)}$ to $\mathbb{B}$. The partial order relation for these functions are defined as:

$$S_1 \rightarrow b_1 \sqsubseteq_{PT} S_2 \rightarrow b_2 \iff S_1 \subseteq S_2 \land b_1 \sqsubseteq b_2.$$ 

The and and or operations are the join and meet operations on this lattice. This is nice since we know from theorem 2.20 when to expect fixed points to exist. Since the sets $2^{CL(\Phi)}$ and $\mathbb{B}$ are both finite, the set of functions $2^{CL(\Phi)} \rightarrow \mathbb{B}$ is also finite. Thus they form a complete lattice.

The algorithm calculates one property transformer for each pair of stack symbol and variable in the simple form of the property.

If we have calculated all these functions, and we know which $\Psi_i \in \Delta_\epsilon \subseteq CL(\Phi)$ that are true for the empty process, the process where the stack is empty, and no transitions are possible, we can calculate if s state $\alpha \in V^*$ fulfills $\Phi$ by simply evaluating the function

$$[\alpha]^\Psi_1(\Delta_\epsilon).$$

Where $\Psi_1$ is the root of the property. If $\alpha$ is a string of stack symbols longer than one, the property transformers of the individual stack symbols have to be combined using the following rule:

$$[A\beta]^\Psi(\Delta) = [A]^\Psi(\{\Upsilon \in CL(\Phi) | [\beta]^\Upsilon(\Delta) = 1\}).$$

(4.2)

Where $\alpha = A\beta$. The purpose of the rule is to execute the first property transformer on the precondition of the following ones. If $\beta$ is longer than 1, the rule has to be applied again to $\beta$.

### 4.2 Property transformers

The main problem of the algorithm is to calculate the property transformers. The transformers are defined in table 4.1, though the definition requires some explaining, or thinking, to make sense. First of all, we need a valuation $V$ which assigns values to the variables. This valuation is updated continuously along the way of the computation. The six rules on top of the table is fairly simple. A transformer of an atomic proposition is either true or false depending on if the proposition is true for the stack symbol of the transformer. A property transformer of a variable is true if the valuation of that variable is true for the stack symbol. A transformer $[A]^\Psi_1 \land \Psi_2$ for a disjunction is true if the transformer of both $[A]^\Psi_1$ and $[A]^\Psi_2$ are true. Similarly for conjunction. A property transformer $[A]^{[a]}\Psi_1$ is true if all $a$ transitions from $A$ renders a state where $\Psi_1$ is true. The case $[A]^{(a)}\Psi_1$ is true if at least one $a$ transition form $A$ renders a true state.
\[ [A]^{\mu}_{V} = \lambda(\Delta).1 \quad \quad [A]^{\nu}_{V}^{\psi_{1},\psi_{2}} = [A]^{\psi_{1}} \sqcup [A]^{\psi_{2}} \]
\[ [A]^{\nu}_{V}^{\psi} = \mu X.\Psi \quad \quad [A]^{\mu \psi}_{V} = \nu X.\Psi \]

\[ [A]^{\mu X,\psi}_{V} = \text{sel}_{A}\left(\sqcup \left\{ (h_{A_{1}}, \ldots, h_{A_{n}}) | \forall 1 \leq i \leq n [A]^{\psi}_{V}(X,A) \rightarrow h_{A_{i}}^{X}, i \leq j \leq n \right\} \right) \]
\[ [A]^{\nu X,\psi}_{V} = \text{sel}_{A}\left(\sqcap \left\{ (h_{A_{1}}, \ldots, h_{A_{n}}) | \forall 1 \leq i \leq n h_{A_{i}}^{X} \sqsubseteq [A]^{\psi}_{V}(X,A) \rightarrow h_{A_{i}}^{X}, i \leq j \leq n \right\} \right) \]

\[ [c]^{\psi}_{V}(\Delta) = \text{mem}_{\Psi}(\Delta) \]
\[ [A\alpha]^{\psi}_{V}(\Delta) = [A]^{\psi}_{V}((\{ \gamma \in CL(\Phi) | [\alpha]^{V}_{\gamma}(\Delta) = 1 \}) \]

**Table 4.1. Property transformers**

The last two rules of table 4.1 is dealing with the empty and the sequentially composed process. A property transformer on the form \([c]^{\psi}_{V}\) will appear when there are sub formulas like \([a]^{\psi}_{V}\) and there is a transition \(A \xrightarrow{\alpha} \epsilon\) in \(C\). In the control flow problem this happens when the sub formula looks like \([m \text{ ret}]^{\psi}_{V}\). The function \(\text{mem}_{\Psi}\) is defined as:

\[
\text{mem}_{\Psi}(\Delta) \triangleq \begin{cases} 
1 & \text{iff } \Psi \in \Delta \\
0 & \text{otherwise}.
\end{cases}
\]

This means that \(\Psi\) is in the precondition condition of \([c] \) if it is in the post condition.

Since \(\mu\)-calculus has fixed point operators to deal with recursion, we need a way to calculate \([A]^{\mu X,\psi}_{V}\) and \([A]^{\nu X,\psi}_{V}\). The definition of the property transformers of the fixed point operations are not nearly as nice as the ones for the other operations. Even with the information that the function \(\text{sel}_{A}\) selects \(A\) from the resulting tuple, the definition is still very difficult to understand. Those of you who paid attention in the preliminaries might see some resemblance with theorem 2.20. This is not a coincident. We will not try to explain the exact meaning of the definition, those who are interested can look at it in [2], we will instead give an explanation on how it is computed.
4.2.1 Computing the fixed points

Since we have concluded that the property transformers are elements of a finite lattice and that evaluations of $\mu$-calculus formulas with certain restrictions is monotone we know that evaluation is also continuous. And since evaluation is continuous and the property transformers form a complete lattice we can use the procedure suggested in theorem 2.21. Calculating the property transformer with no interior fixed points is fairly straightforward. If we are computing a property transformer of a minimal fixed point $\mu X.\Psi(X)$, we initiate the variable $X$ in $\Psi$ to 0, and start computing using the update rules of table 4.1. As we proceed with these calculations, the value of $X$ will, possibly, change. When no more changes takes place, the fixed point is reached. If we are looking for a maximal fixed point, we in stead start with initializing the variable to 1, and then follow the same procedure.

A small example is in order here to illustrate the last few sections.

Example 4.1 Simple fixed point computation. We use the sample program from chapter 3 which had a context-free process representation

$$A \xrightarrow{m\hspace{1pt}\text{call}\hspace{1pt}m} A \cdot B, \hspace{1pt}A \xrightarrow{\epsilon} B, \hspace{1pt}B \xrightarrow{m\hspace{1pt}\text{ret}} \epsilon.$$  

We take a simple $\mu$-calculus formula as property

$$\nu X.([m\hspace{1pt}\text{call}\hspace{1pt}m]X \wedge (\epsilon)\text{return}).$$

This property states that it is possible to make a $\epsilon$ action and end up in a state where $\text{return}$ is true, and that any number of $m\hspace{1pt}\text{call}\hspace{1pt}m$ does not change that. This is not a behavioral property, but a structural one, but it is still a nice example of the algorithm for non-nested fixed point computation.

First we put the property formula in simple form, and create the property transformer for each stack symbol sub formula pair.

<table>
<thead>
<tr>
<th>Formula</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Psi_1 \triangleleft \Psi_2$</td>
<td>$[A]^{\Psi_1} \triangleleft [A]^{\Psi_2}$</td>
<td>$[B]^{\Psi_1} \triangleleft [B]^{\Psi_2}$</td>
</tr>
<tr>
<td>$\Psi_2 = \Psi_3 \land \Psi_4$</td>
<td>$[A]^{\Psi_2} = [A]^{\Psi_3} \cap [A]^{\Psi_4}$</td>
<td>$[B]^{\Psi_2} = [B]^{\Psi_3} \cap [B]^{\Psi_4}$</td>
</tr>
<tr>
<td>$\Psi_3 = [m\hspace{1pt}\text{call}\hspace{1pt}m]\Psi_1$</td>
<td>$[A]^{\Psi_3} = [AB]^{\Psi_1}$</td>
<td>$[B]^{\Psi_3} = 1$</td>
</tr>
<tr>
<td>$\Psi_4 = (\epsilon)\Psi_5$</td>
<td>$[A]^{\Psi_4} = [B]^{\Psi_5}$</td>
<td>$[B]^{\Psi_4} = 0$</td>
</tr>
<tr>
<td>$\Psi_5 = return$</td>
<td>$[A]^{\Psi_5} = 0$</td>
<td>$[B]^{\Psi_5} = 1$</td>
</tr>
</tbody>
</table>

Since we are calculating a maximal fixed point we initiate all variables to 1, and start computing. The result is will be as follows.
In order to deal with nested fixed point we initiate variables of least fixed point as 0 and greatest fixed point as 1. Compute the innermost fixed points first and use that result to compute the fixed points outside of the innermost. When this is finished, the result computed for the outer fixed points is used to compute the inner most fixed points again. This goes on until all fixed points are reached.

### 4.3 Evaluating resulting property transformers

When the property transformers have been computed we simply compose the state of the process we wish to compute using sequential composition. Then we apply that property transformer on the set of sub formulas $\Delta$, of $\Phi$ that are true for the empty process. This can easily be done using the method described above. Since there are no transitions in the empty process there will be no property transformers containing the $\text{mem}_k$ function. To check if a sub formula $\Psi_i$ is in the pre condition of the empty process we simply check if $[\epsilon]^{\Psi_i} = 1$.

Other similar model checkers are able to produce counter-examples when the property does not appear to hold. This is not the case with this algorithm, or at least not that we know of. One problem of finding a counter example is that they may be infinite.

### 4.4 A detailed example

In table 4.2 we give a larger example that shows most details of the algorithm. The example is taken from [2], but here we have carried out and presented all the computations.
<table>
<thead>
<tr>
<th>Context-free process</th>
<th>Property formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \xrightarrow{a} AB$</td>
<td>$\mu X.\nu Y.([b]X \land [a]Y)$</td>
</tr>
<tr>
<td>$A \xrightarrow{b} e$</td>
<td></td>
</tr>
<tr>
<td>$B \xrightarrow{b} e$</td>
<td></td>
</tr>
</tbody>
</table>

### Simple formula

| $\Psi_1 \triangleq \Psi_2$ | $[A]_{\Psi_1} \triangleq [A]_{\Psi_2}$ | $[B]_{\Psi_1} \triangleq [B]_{\Psi_2}$ |
| $\Psi_2 \triangleq \Psi_3$ | $[A]_{\Psi_2} \triangleq [A]_{\Psi_3}$ | $[B]_{\Psi_2} \triangleq [B]_{\Psi_3}$ |
| $\Psi_3 = \Psi_4 \land \Psi_5$ | $[A]_{\Psi_3} = [A]_{\Psi_4} \land [A]_{\Psi_5}$ | $[B]_{\Psi_3} = [B]_{\Psi_4} \land [B]_{\Psi_5}$ |
| $\Psi_4 = [b]_{\Psi_6}$ | $[A]_{\Psi_4} = [b]_{\Psi_6}$ | $[B]_{\Psi_4} = [b]_{\Psi_6}$ |
| $\Psi_5 = [a]_{\Psi_7}$ | $[A]_{\Psi_5} = [a]_{\Psi_7}$ | $[B]_{\Psi_5} = [a]_{\Psi_7}$ |
| $\Psi_6 = \Psi_1$ | $[A]_{\Psi_6} = [A]_{\Psi_1}$ | $[B]_{\Psi_6} = [B]_{\Psi_1}$ |
| $\Psi_7 = \Psi_2$ | $[A]_{\Psi_7} = [A]_{\Psi_2}$ | $[B]_{\Psi_7} = [B]_{\Psi_2}$ |

| $A_{\Psi}$ | $A_{\Psi}^2$ | $A_{\Psi}^3$ | $A_{\Psi}^4$ | $A_{\Psi}^5$ | $A_{\Psi}^6$ | $A_{\Psi}^7$ | $B_{\Psi}$ | $B_{\Psi}^2$ | $B_{\Psi}^3$ | $B_{\Psi}^4$ | $B_{\Psi}^5$ | $B_{\Psi}^6$ | $B_{\Psi}^7$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 1     | 1     | 0     | 0     | 1     | 1     | 0     | 0     | 1     | 0     | 1     | 0     | 1     |
| 1     | 1     | $\Psi_6$ | 1     | 0     | 0     | 1     | $\Psi_6$ | 1     | 0     |
| 1     | $\Psi_6$ | $\Psi_6$ | 1     | 0     | 1     | $\Psi_6$ | 1     | 0     | 1     |
| $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | 1     | 0     | 0     | $\Psi_6$ | 1     | 0     | $\Psi_6$ |
| $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | 1     | 0     | $\Psi_6$ | 1     | 0     | $\Psi_6$ |
| $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | 1     | 0     | $\Psi_6$ | 1     | 0     | $\Psi_6$ |
| $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | 0     | $\Psi_6$ | 1     | 0     | $\Psi_6$ |
| $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | 0     | $\Psi_6$ | 1     | 0     | $\Psi_6$ |
| $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | 0     | $\Psi_6$ | 1     | 0     | $\Psi_6$ |
| $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | $\Psi_6$ | 0     | $\Psi_6$ | 1     | 0     | $\Psi_6$ |

### Table 4.2

All the details of an execution of the algorithm.
4.5 Correctness and complexity

Since these computations are continuous and on finite domains they will terminate, and theorem 2.21 suggests that they will produce the correct result, but a formal proof that this is the case requires advanced logic and domain theory. We do not discuss this here and refer the interested reader to [2] for a detailed proof.

The complexity of the algorithm is

\[ O(|\Phi| \times (|C| \times 2^{|\Phi|})^{ad(\Phi)+1}). \]

where \(|\Phi|\) is the size of the property to prove, the number of equations in the simple form of the property, \(||C||\) is the size of the process, i.e. the number of symbols in the alphabet. Finally, \(ad(\Phi)\) is the alternation depth of the property formula.

4.6 Observations

As we can see from the complexity analysis, the running time increases dramatically with larger formulas. From this observation, a simple modification of the generation of context-free processes described in section 3.3 can be proposed. Without losing much expressive power, we can alter the model building rules to.

\[ v \rightarrow v' \quad v \models \neg \text{return} \]

\[
\frac{v}{\epsilon} \quad v' \quad v \models \neg \text{return}
\]

(4.3)

\[ v_1 \xrightarrow{m_2} v_1' \quad v_1 \models \neg \text{return} \quad v_2 \models m_2 \wedge \text{entry} \]

\[
\frac{v_1 \xrightarrow{m_2} v_2 \cdot v_1'}{v_1 \xrightarrow{m_2} v_2 \cdot v_1'}
\]

(4.4)

\[ v_1 \models \text{return} \quad v_1 \models m_1 \]

\[
\frac{v_1 \xrightarrow{\text{ret}} \epsilon}{v_1 \xrightarrow{\text{ret}} \epsilon}
\]

(4.5)

This gives much smaller formulas, since \(m_1\) call \(m_2\) and \(m_1'\) call \(m_2\) are assigned the same label. Since we are not often interested in who is calling, rather where the call goes, we often benefit from this simplification. We can
make an even bolder simplification in replacing all labels with just one single, $\epsilon$ for example.

$$v \rightarrow v' \quad v \models \neg \text{return}$$

$$v \xrightarrow{\epsilon} v'$$  \hfill (4.6)

$$v_1 \xrightarrow{m_2} v_1' \quad v_1 \models \neg \text{return} \quad v_2 \models m_2 \land \text{entry}$$

$$v_1 \xrightarrow{\epsilon} v_2 \cdot v_1'$$  \hfill (4.7)

$$v_1 \models \text{return} \quad v_1 \models m_1$$

$$v_1 \xrightarrow{\epsilon} \epsilon$$  \hfill (4.8)

Making this simplification does not decrease the expressive power since it is always possible to make the same statement as before.

$$[m_1 \text{ call } m_2] \Phi \equiv \neg m_1 \lor [m_2] \Phi \equiv \neg m_1 \lor \text{return} \lor \left[ \epsilon \right](\neg m_2 \lor \Phi)$$  \hfill (4.9)

$$[m_1 \text{ ret}] \Phi \equiv \neg m_1 \lor [\text{ret}] \Phi \equiv \neg m_1 \lor \neg \text{return} \lor (\text{return} \land [\epsilon] \Phi)$$  \hfill (4.10)

In the new representation the Always pattern will change to:

$$\text{Always } \Phi = \nu Z. \Phi \land [\epsilon] Z$$  \hfill (4.11)

Which has size $|\text{Always } \Phi| = |\Phi| + 3$ in stead of the original which is of size $|\text{Always } \Phi| = |\Phi| + 2 + |\text{Act}|$. Where $|\text{Act}| \leq |M|^2 + |M| + 1$ is limited by the number of possible method calls, plus the number of returns, plus the $\epsilon$ action.

The notion of reachability given in equation 3.7 will be altered to

$$\neg m \lor \nu X.(\neg m' \land [\epsilon] X).$$  \hfill (4.12)

**Proposition 4.2.** All labels generated by the construction procedure described in [5] and chapter 3 can be replaced by one single action $\epsilon$, without loss of generality.

This change of the model does not only effect the running time. It also drastically decrease the amount of memory used.

From the example runs of the algorithm it appears that to compute the next state of the property transformers we need to store the old states. Due to monotonicity of the update procedure, this is not the case.

**Proposition 4.3.** Property transformer updates can be done in place.
Chapter 5

Implementation

During our work on the implementation a few interesting issues were uncovered and solved. In this chapter we will give a presentation of those that we found interesting.

5.1 Property transformer representation

The property transformers are boolean functions in $|\Phi|$ variables. What we essentially need is a data structure that can store and manipulate expressions like $\Psi_1 \land (\Psi_2 \lor \Psi_3)$. Apart from the usual boolean operation, we also need to be able to replace a variable with an expression.

Example 5.1 Variable substitution. A simple example of variable substitution:

$$\Psi_1 \land (\Psi_2 \lor \Psi_3) \rightarrow [\Psi_2 = \Psi_1 \lor \Psi_4, \Psi_3 = \Psi_2 \lor \Psi_4] \rightarrow \Psi_1 \land (\Psi_2 \lor \Psi_4).$$

We also need to be able to compare two expressions for equality.

Fortunately there is a data structure that solves most of these problems, as well as it is a very compact structure, ordered binary decision diagrams, OBDD. We have chosen to use a library called CUDD (Colorado University Decision Diagrams) which is a C/C++ package for manipulating ordered binary, arithmetic and algebraic decision diagrams. One nice property of OBDDs is that they are canonical, and thus equality checking is easy. The standard boolean operations are atomic in OBDD so they don’t cause trouble either. In CUDD there is also functionality for simultaneous substitution like in example 5.1.
5.2 Calculating the transformers

To calculate the transformers we build a tree of fix point dependencies. A fix point transformer on formula $\Psi_f$ is depending on a sub formula $\Psi_i$ if $\Psi_i$ is captured inside $\Psi_f$ and not captured inside another fix point.

Now we move on to building rules for updating all other transformers. For each fixed point we then calculate first its interior fix points, then the other sub formulas. If a fixed point is not reached, i.e. something has changed during the computation, the process is repeated.

**Example 5.2 Fixed point dependencies.** The formula $\mu X. (\nu Y. ([e]X \land [Act]Y) \lor \nu Z. (\langle e \rangle Z))$ which has the simple form

$$\begin{align*}
\Psi_0 &\doteq \Psi_1 \\
\Psi_1 &\doteq (\Psi_2 \lor \Psi_3) \\
\Psi_2 &\doteq \Psi_4 \\
\Psi_3 &\doteq \Psi_{10} \\
\Psi_4 &\doteq \Psi_7 \land \Psi_5 \\
\Psi_5 &\doteq [Act]\Psi_8 \\
\Psi_6 &\doteq \Psi_0 \\
\Psi_7 &\doteq [e]\Psi_6 \\
\Psi_8 &\doteq \Psi_2 \\
\Psi_{10} &\doteq \langle e \rangle \Psi_{11} \\
\Psi_{11} &\doteq \Psi_3.
\end{align*}$$

Has the fixed point dependencies $dep(\Psi): CL(\Phi) \rightarrow 2^{CL(\Phi)}$:

$$\begin{align*}
dep(\Psi_0) &= \{\Psi_1, \Psi_2, \Psi_3\} \\
dep(\Psi_2) &= \{\Psi_4, \Psi_5, \Psi_6, \Psi_7, \Psi_8\} \\
dep(\Psi_3) &= \{\Psi_{10}, \Psi_{11}\}.
\end{align*}$$

As a result of proposition 4.3, once we have calculated a new value for a property transformer, we can immediately use it, and discard the old value.

**Algorithm 5.3 Calculating the transformers.** The algorithm to calculate the transformers is fairly simple.

1. Calculate transformer dependencies.
2. Start at the outermost level of fix points.
3. Compute value of each interior fix point, by initiating the fixed point variable to $\bot$ for least fix points and $\top$ for greatest fix points. Then use step 3 to 5 recursively on the interior fix points.
4. Compute value of fix points on this level, using the value of the fixed point from 3.
5. If the value of the current fix point value has not changed, a fixed point is reached.
5.3 Implementation using CUDD

As mentioned before we use a code library called CUDD. We used only a fraction of CUDDs capabilities, essentially only the diagram variable type, and the two operations logical and and logical or. Following the instructions algorithm 5.3 each $\Psi$ is represented by a CUDD decision diagram variable and the next value of $\Psi_i$ is calculated directly from the values of the $\Psi$ on the other side of the equality. From examples 5.4 and 5.5 along with algorithm 5.3 you can see the main features of the property transformer update procedure.

*Example 5.4 Simple CUDD example.*

\[
\Psi_7 = \Psi_5 \lor \Psi_6
\]

is implemented

01 BDD $[x]^{\Psi_i}$

02 for each symbol $a$

03 $[a]^{\Psi_7} = [a]^{\Psi_5} + [a]^{\Psi_6}$

*Example 5.5 CUDD example of productions.*

\[
\Psi_7 = [e]^{\Psi_6}
\]

is implemented

01 BDD $[x]^{\Psi_i}$

02 $\Psi_7 = \text{CuddManager.dbbOne}()$

03 for each symbol $a$

04 $[a]^{\Psi_7} = \text{CuddManager.dbbOne}()$

05 for each symbol $b$

06 for each $e$ transitions from $a$ to $b$

07 $[a]^{\Psi_7} = [a]^{\Psi_7} \cdot [b]^{\Psi_6}$

5.4 Possible improvements on the implementation

There are a few things that might be possible to try in order to get more out of the algorithm.
The property transformer computations can be performed on parallel processors if a small amount of overhead to handle communications of results is added. This is possible since fix points on the same level are never dependent on each other, apart from that their results all add up to the outer fix points values, which may propagate down again. As will be apparent from the performance measurements in the next section, a run of this algorithm can be rather time consuming, and the use of parallel computers may be interesting.

Another possible improvement is standard code optimization. Since the focus of the implementation was to test the algorithm and focus has been placed on making the implementation readable, there might be many small coding details to tune in order to gain speed.
Chapter 6

Practical Evaluation

The final part of this thesis concerns the performance and reliability of the implementation and the algorithm itself. The algorithm is apparently an answer to the problem posed in chapter 3, but since we are interested in using the algorithm there are a few more concerns.

6.1 Correctness

It's always an important question if a program delivers a correct result upon termination. Specially so when it comes to programs who proves correctness of other programs. The algorithm is proved to be correct in [2]. The model building scheme of chapter 3 is well studied by the verification community, but not the modifications made in section 4.6. The question is, have we implemented the algorithm and surrounding tools correctly?

The implementation of the algorithm is a bit too complex to be completely bug free. But we have written both unit and system tests in order to find and eliminate the errors. The tests consists of a large number of tests on the property transformers, testing the three main operations, meet, join and composition. There are also several tests that runs the whole algorithm on examples with known results.

We do not encourage use of the implementation unless a more thoroughly investigation of the correctness is done. We feel confident that the implementation works on most real world data, but there are probably a few bugs, that on certain data, makes the program deliver incorrect output or no output at all.
6.2 Performance

To test the performance of the algorithm we have verified properties about the applets studied in [5] on a 1000MHz Pentium with 256MB RAM running Linux.

In [5] Huisman et al described how to make a model of a JavaCard application called Gemplus PACAP. The Gemplus is formed of three different types of applets. One CardIssuer, one Purse and several Loyalty. Huisman et al only studied the Purse and Loyalty applets. A card contains a Purse and one Loyalty applet for each loyalty program the card holders is part of. More Loyalty applets will be added as the card holder joins more programs. The Purse and Loyalty applets interchange data so the Loyalty applets can award credits for purchases made. Huisman et al made a model and constructed properties to show that no illicit interaction could occur between the applets on a card. We have taken the model of a Purse and a Loyalty applet and run some tests on them using one of the properties presented in [5].

We have used the model checking tool to verify a property on the form

\[ \text{Within m (CanNotCall M \{m\})}. \]

We tested the algorithm both on the original formulation of the context-free process construction rules in chapter 3 and on the two modifications suggested in section 4.6. The size of the context free process tested was 2788 symbols using the method described in 3.3, and the size of the formula was 137139 using the original way of constructing formulas, 1229 when using the reduction first suggested in 4.6, and finally the size of 8 when using proposition 4.2. The results are presented in table 6.1.

We soon discovered that one important limitation on the performance of the algorithm was memory usage. The memory needed is proportional to the number of symbols in the context-free process multiplied with the size of the property formula. Large examples quickly consumed the all free memory, and the computer needed to use disc memory, which is much slower than usual RAM. The memory consumption for the largest example was so big that the allocation of that much memory failed, and the computation failed to deliver a result. In order to get a result we had to reduce the problem. We used the result presented in the end of chapter 4. Even with some of these modifications the memory consumption was huge. To see how the algorithm performs when not suffering from lack of memory we have also measured the processor time used by the process running the algorithm. A large example that consumes much more memory than available will have to wait for the system to swap, as suggested on lines two and three in table 6.1 if there was enough memory for the computations to be done entirely in RAM the running time will decrease drastically.
The results of the test runs are presented in table 6.1.

To verify that it really is the memory that limits the usability of the implementation we constructed a number of examples with size, both of the system and of the properties, between the $\epsilon$ example on line three and the example on line two. The test cases presented in table 6.2 were constructed using the test case on line three and successfully adding symbols, productions and connectives to the system and the property. This was done without consideration to what the new runs gave as result, but only to test the running time of the algorithm.

As you can see from table 6.2 the amount of memory consumed is fairly proportional to the size of the formula times the size of the process. This is expected, the algorithm requires a property transformer for each formula element, process symbol pair. You can also see that the memory consumption is quite large on average. Remember, these are small examples.

There are two direct solutions to the memory problem. A more memory efficient implementation of the algorithm, or a more powerful computer, with more RAM. It's highly probable that a substantial improvement on the memory consumption is possible, if some work is put into optimizing the usage. Neither of these are fully satisfactory since they won't give us the solution to a problem of the original size and formulation. The proper solution appears to be to use some reduction to make the problem easier to solve. The reduction mentioned in proposition 4.2 is one possible solution.
| $|C|$ | $\Phi$ | Memory | Real time |
|---|---|---|---|
| 137 | 8 | 8MB | 10s |
| 513 | 8 | 26MB | 27s |
| 1032 | 8 | 62MB | 57s |
| 2788 | 8 | 107MB | 1m 35s |
| 137 | 16 | 31MB | 47s |
| 513 | 16 | 63MB | 1m 15s |
| 1032 | 16 | 117MB | 2m 7s |
| 2788 | 16 | 232MB | 3m 38s |
| 137 | 32 | 57MB | 1m 15s |
| 513 | 32 | 101MB | 2m 1s |
| 1032 | 32 | 213MB | 4m 13s |
| 2788 | 32 | 410MB | 19m 28s |
| 137 | 64 | 81MB | 1m 37s |
| 513 | 64 | 130MB | 2m 22s |
| 1032 | 64 | 178MB | 5m 24s |
| 2788 | 64 | 617MB | 31m 32s |
| 137 | 128 | 105MB | 2m 50s |
| 513 | 128 | 173MB | 4m 31s |
| 1032 | 128 | 201MB | 7m 31s |
| 2788 | 128 | 673MB | 38m 7s |
| 137 | 256 | 117MB | 3m 12s |
| 513 | 256 | 187MB | 5m 31s |
| 1032 | 256 | 267MB | 16m 58s |
| 2788 | 256 | 679MB | 54m 31s |
| 137 | 511 | 143MB | 4m 48s |
| 513 | 511 | 201MB | 6m 19s |
| 1032 | 511 | 313MB | 34m 34s |
| 2788 | 511 | 712MB | 1h 4m 49s |
| 137 | 1024 | 169MB | 5m 39s |
| 513 | 1024 | 231MB | 7m 1s |
| 1032 | 1024 | 399MB | 41m 17s |
| 2788 | 1024 | 798MB | 1h 23m 14s |
| 137 | 1530 | 213MB | 6m 31s |
| 513 | 1530 | 396MB | 36m 39s |
| 1032 | 1530 | 721MB | 1h 3m 1s |
| 2788 | 1530 | - | - |

**Table 6.2.** Time and memory consumption on test runs of the algorithm.
Chapter 7

Conclusions

The most important result of this thesis is that the algorithm presented in chapter 4 solves the problem posed in chapter 3. Using proposition 4.2 it even solves it in a reasonable amount of time and with reasonable amount of resources. With some effort put into optimizing the performance, both in time and memory consumption, the implementation should be usable, given that it is correct.

7.1 Future work

There are several aspects of this thesis that may be of interest to look further into.

- Verification of the correctness of the implementation.
- Performance tuning, perhaps even parallelizing the implementation.
- Analysis of the implications made in proposition 4.2.
- It would be nice if it was possible to get more information out of the algorithm, such as a counterexample when a formula is found not to hold. This is in general not possible, since the counterexamples could be of infinite size.
- It is possible that a more efficient algorithm exists. If this shows to be an important kind of problem to solve, it might be interesting to investigate the existence of a different algorithm.
- The memory problem presented in chapter 6 must be addressed, if this implementation should be used for real world examples.
References


