(Unterschrift Betreuer)

TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

# Alternation as a programming paradigm

ausgeführt am
Institut für Informationssysteme
der Technischen Universität Wien

unter der Anleitung von
Prof. Dr. Reinhard Pichler
&
Privatdoz. Dr. Stefan Woltran

durch
WOLFGANG DVOŘÁK
Floridusgasse 2-10/3/3
1210 Wien

Wien, am 10. 02 2009

(Unterschrift Verfasser)

**Erklärung zur Verfassung der Arbeit**

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, am 10.02.2008

Wolfgang Dvořák

**Abstract**

In computational complexity theory alternating machines are nondeterministic machines which alternate between existential and universal states. A nondeterministic machine in each computation state may have several possibilities for the further computation. An existential state accepts an input if at least one of this possible computations leads to acceptance. An universal state accepts an input only if all possible computations lead to acceptance.

There is a strong connection between the complexity classes defined by alternating machines and deterministic complexity classes. So the concept of alternating algorithms is often used in computational complexity theory to show that a problem is decidable in a specific complexity class. Alternating algorithms are useful to solve problems in theory but there is no language to use this concept for practical programming.

Very natural examples of problems with an alternating character are games, more precisely, the question if there is a winning strategy for the start player in the game. To find a winning strategy we recursively look for a move for the start player such that for all possible moves of the opponent he has a winning strategy. Many popular games, for example Go, are known to be inherent hard problems for computational purposes, so that they can't be solved in practice. But there is also a large class of games that can be computed in reasonable time.

What is missing is a framework that allows us to apply this useful theoretic alternating constructs in practical programming. Further it should simulate the alternating programs in a feasible way to make alternating programming practicable.

In this thesis we first identify natural alternating problems which are feasible computable. So these are problems which induce very complex deterministic algorithms but can be intuitively solved with an alternating algorithm.

Then we break down the theoretic model of alternating machines to an intuitive language for writing alternating programs in practice. We confirm the intuitiveness of our language by presenting some alternating programs. Further we introduce a framework that simulates these alternating programs in an efficient way.

**Zusammenfassung**

In der Komplexitätstheorie versteht man unter alternierenden Maschinen, nichtdeterministische Maschinen bei denen existentielle und universelle Zustände alternieren. Unter einer nichtdeterministischen Maschine versteht man eine Maschine bei der es in jedem Zustand mehrere Möglichkeiten für die weitere Berechnung geben kann. Existentielle Zustände akzeptieren die Eingabe dann wenn mindestens eine der möglichen Berechnungen die Eingabe akzeptiert. Universelle Zustände akzeptieren die Eingabe nur dann wenn alle möglichen Berechnungen die Eingabe akzeptieren.

Die Komplexitätsklassen die mittels alternierenden Maschinen definiert werden stehen in einem starken Zusammenhang mit der Hierarchie der gewöhnlichen deterministischen Komplexitätsklassen. Deshalb werden alternierende Algorithmen oft verwendet um zu zeigen das Probleme in bestimmten Komplexitätsklassen lösbar sind. Alternierende Algorithmen sind also ein nützliches Werkzeug in der Komplexitätstheorie, dennoch gibt es keine Programmiersprache die dieses Konzept für die Praxis bereitstellt.

Das Paradebeispiel für Problem mit einem alternierenden Kern sind Spiele. Die Frage die uns bei Spielen interessiert ist ob es eine Gewinnstrategie für den Startspieler gibt. Um diese Frage zu beantworten müssen wir rekursiv folgendes Problem lösen. Existiert ein Zug für den Startspieler sodass für alle möglichen Züge des Gegners es wieder eine Gewinnstrategie für den Startspieler gibt. Viele bekannte Spiele, wie zum Beispiel Go, sind in der Komplexitätstheorie als inhärent schwierige Probleme bekannt und damit auch für praktische Programmierung uninteressant. Es gibt aber eine weite Bandbreite von Spielen und anderen alternierenden Problem mit niedriger Komplexität die auch in der Praxis berechnet werden können.

Alternierung scheint also ein nützliches Programmier Paradigma in der Theorie zu sein, es stellt sich also die Frage ob es sich auch in einer praktischen Programmiersprache umsetzen lässt. Um es in der Praxis auch nutzen zu können stellt sich weiter die Frage nach einer Umgebung in der sich die alternierenden Programme auch effizient ausführen lassen.

In dieser Arbeit identifizieren wir zunächst natürliche alternierende Probleme die mit akzeptablem Aufwand berechenbar sind. Das sind Probleme die sehr komplizierte deterministische Algorithmen benötigen aber mit dem Konzept der Alternierung sehr intuitiv gelöst werden können.

Dann führen wir eine Sprache ein die es ermöglicht in einer kompakten und intuitiven Art alternierende Programme zu schreiben. Wir untermauern die Natürlichkeit dieser Sprache mit einer Reihe von Programmbeispielen für die vorher identifizierten Probleme. Weiters stellen wir ein System vor, das diese alternierenden Programme auf eine effiziente Art simuliert.

# Contents

# Chapter 1

# Introduction

Alternating problems, problems where existential and universal states alternate, are well studied problems in computational complexity theory, but there are no programming languages to write alternating algorithms in practice. In this thesis, we introduce a Java language extension for alternating algorithms based on the concept of Alternating Turing machines by [Chandra et al., 1981].

An alternating machine is a special nondeterministic machine, where every configuration may have many successors and the states of the machine are divided in existential states, universal states and the halting states accept and reject. An existential state accepts an input if at least one of the successors accept. A universal state accepts only if all the successors accept.

Chandra et al., 1981 have shown that an alternating Turing machine can be simulated by a deterministic Turing machine and therefore it is possible to simulate an alternating machine with a traditional computer. Further, they have show that the hierarchy of the main complexity classes of alternating Turing machines corresponding to the hierarchy of complexity classes of deterministic Turing machines shift by one level. Although alternating Turing machines seem to be very unnatural models of computation they are often used in complexity theory to show some problems to be in or complete for some complexity class. There are many problems which have intuitive alternating algorithms, for example, Boolean circuit evaluation, finding a winning strategy in games and logic problems on Horn clauses.

A very natural alternating problem is the question if there is a winning strategy in a two-player game. Searching for a winning strategy, we ask if there exists a move for Player 1 so that for all possible moves of Player 2 there is a winning strategy for Player 1. This is a very simple algorithm but there is no easy way to notate these existential and universal quantifiers in Java or any other imperative programming language.

So what we desire is a programming language that offers constructs to write alternating programs in an intuitive way. In such a language the programmer should specify how to compute the successors and the quantifier of a state, and the corresponding framework has to evaluate the alternating program. Further the framework simulating the alternating machine has to work efficiently to make alternating programming practicable.

One of the main problems with alternating algorithms is the, in general exponentially large computation tree. This leads in worst case to an exponential time simulating algorithm. If we implement polynomial time alternating Turing machines without restrictions we will get a simulation in PSPACE. Coming back to the problem of winning strategies in games, the problem of deciding if there is a winning strategy in chess or Go is well known to be an inherent hard problem.

Another problem is caused by alternating machines with a computation path that does not halt, in games this means that there is a cycle in the strategy. There is a result in Chandra et al., 1981 how to construct an equivalent machine such that all computation paths halt by introducing a counter, but using this technique produces a polynomial blow up of space requirement for the simulation. This has no effect on the theoretic usage but it is indeed infeasible for many practical applications.

To prevent exponential space requirements and running time caused by the exponentially big computation tree, we either have to restrict the size of the computation tree or the workspace used in the alternating machine. The former would lead to the complexity class of LOGCFL Venkateswaran, 1987 while the latter leads to a class of alternating machines equivalent to deterministic polynomial time Chandra et al., 1981. The alternation extension in the thesis follows the idea of space restriction to capture a much larger class of alternating problems.

Eliminating non halting computation paths in the usual way,i.e. with a counter which increases each computation step and rejects if exceeding a polynomial bound, we get a polynomial blow up of space requirements for the simulation algorithm. To avoid this blow up, we shift the detection of non-halting paths in alternating programs from the programming level to the simulation algorithm. So our framework handles cycles in programs and the programmer need not care about this when writing an alternating program.

In this thesis we introduce a Java language extension that offers alternating constructs to write alternating programs in a very intuitive way. This extension only adds a few new constructs to ordinary Java and therefore is quite intuitive and easy to learn. The main advantages of this alternating programming language are much shorter programs than in traditional Java, the very intuitive manner they are written, and therefore, a well readable code. The efficient simulation algorithm in the according framework ensures that these programs can be evaluated in reasonable space and time bounds. Further we give examples of programs and problems that demonstrate the existence of interesting and feasible alternating problems and beside problems in PSPACE and beyond.
In this thesis we introduce

- a sample of natural alternating problems,
- a Java language extension that gives the programmer the facility to write alternating algorithms,
- a tabled evaluation program simulating an alternating machine in an efficient way.

## Organisation of the Thesis

In Chapter 2 we first give an overview of the basic concepts in computational complexity theory. We present a formal concept of alternation, the alternating Turing machine. The definitions of alternating complexity classes and their connection to the deterministic complexity hierarchy are listed. Furthermore we identify problems which can be solved intuitively with an alternating algorithm.

In Chapter 3 we introduce both a language and a framework for writing alternating programs in practice. There is a section describing the language, a section that gives example programs and a section that discusses the algorithms used in the framework.

In Chapter 4 we present some empirical tests that compare programs from our framework with ordinary Java programs. Furthermore we compare different algorithms for simulating alternating machines.

In Chapter 5 we summarize our results and give an outlook of possible future work. In the appendix, there is the developed grammar for the alternation language and the Javadoc documentation of the alternation framework.

# Chapter 2

# Computational Complexity

The concept of alternating algorithms we use stems from computational complexity theory. So at the beginning of this chapter there is a short introduction to the basic concepts of complexity theory like Turing machines, reductions, time and space complexity. We introduce these concepts following the book by Papadimitriou, 1994 sometimes we may use slightly modified definitions.

Then we give a formal definition of an alternating Turing machine and a summary of the most important theoretic results in this context. At the end of the section we look at some well-known problems, which can be solved intuitively with an alternating machine.

## 2.1 Machine Models

In theoretical computer science we are interested in formally proving if a problem is solvable by a computer or not. Further if the problem is computable we are interested in the time and space requirements. To formally prove such properties, we need exact definitions of computability, time and space demand. We do this by defining some abstract automatas and look at the problems solvable by these machines. A very common model for computation is the formal model of Turing machines, named after Alan Turing, who introduced this abstract machine model Turing, 1936.

### 2.1.1 Deterministic Turing Machine

> *Despite its weak and clumsy appearance, the Turing machine can simulate arbitrary algorithms with inconsequential loss of efficiency.*Papadimitriou, 1994

A (deterministic) Turing machine is a formal model of a machine that operates on a tape with symbols written on it. This machine has a finite state register and a read/write head to read and modify tape symbols. In every computation step the machine reads the symbol under the tape head and then it does some computation depending on the state stored in the register and the read symbol. In this computation, the machine may change its state, override the tape symbol and move the tape head to a neighbor symbol. So the program of such a machine is a set of rules that tells the machine, for given a state and a symbol, which is the next state, the new tape symbol and how to move the tape head.

There are many slightly different definitions of Turing machine in the literature which are equivalent to each other. Here we give a formal definition what we mean by a Turing machine in this thesis.

**Definiton 2.1.1.** *A (deterministic) Turing machine (TM) is a tuple*

$$M = \langle S, \Sigma, \delta, s \rangle$$

*with:*

- *$S$ a finite set of states - not including the accepting state "yes", the rejecting state "no" and the halting state $h$*
- *$\Sigma$ the alphabet of $M$ - a finite set of symbols including the blank symbol $\sqcup$ and the start-symbol $\triangleright$*
- *$\delta$ a transition function :*

$$\delta : S \times \Sigma \to S \cup \{\text{"yes"}, \text{"no"}, h\} \times \Sigma \times \{\to, \leftarrow, -\}$$

  *The last set denotes the possible cursor movements: $\to$ (right), $\leftarrow$ (left), $-$ (stay)*

- *$s \in S$ the initial state*

The (partial) function $\delta$ can be seen as the program of the Turing machine as it computes a new state, a new symbol and the tape cursor movement for a given state and symbol. Looking at $\delta(s_1, \sigma_1) = (s_2, \sigma_2, D)$ with $s_1, s_2 \in S$, $\sigma_1, \sigma_2 \in \Sigma$ and $D \in \{\to, \leftarrow, -\}$. The function $\delta$ maps the current state $s_1$ and the symbol $\sigma_1$ under the cursor to a new state $s_2$, a new symbol $\sigma_2$ and cursor movement $D$.

The usage of the start symbol is to prevent the machine to move the tape head out of the tape. For this reason the function $\delta$ is restricted when reading the start symbol. Whenever reading the start symbol, the machine is not allowed to override the start symbol and must move the tape head to the right ($\delta(s, \triangleright) = (s', \triangleright, \to)$).

At the start, the Turing machine is in the initial state $s$ and the tape string is initialized with the start symbol $\triangleright$ followed by a finite string $x \in \Sigma^*$ (the input). Then the machine iterates applying the function $\delta$ to change its state, write a symbol and move the cursor.
We have already mentioned that the cursor will never exceed the left end of the string but there is still the possibility of exceeding the right end of the input. If this happens the cursor just scans the blank symbol $\sqcup$. So we have a tape that is infinite to the right side.

So far, we described how the machine works but what is the "return value" for a given input? $\delta$ is defined for all states in $S$ and all symbols in $\Sigma$ so the machine only stops computing when reaching one of the halting states {“yes”, “no”, h}. It is an important fact that a Turing machine started with some input may never reach a halting state and so loops forever.

If a machine $M$ halts on an input $x$ we define the output $M(x)$ as follows: If $M(x)$ halts in the accepting / rejecting state “yes”/“no” we define $M(x) =$ “yes” / $M(x) =$ “no” and we say M accepts / rejects the input $x$.

If the machine halts in the state $h$ then we define the output as the string $y$ of the machine M and write $M(x) = y$. More precise: After the machine reaches the halting state the string of the machine still starts with $\rhd$ followed by finite string $y$ ending with the symbol before the first $\sqcup$. There may follow some non blank symbols, but they are not part of the output. As it is possible that the machine $M$ never halts on some input $x$ we have an additional notation for this case. We simple write $M(x) = \nearrow$.

*Example* 2.1.2. A Turing machine $M$ that computes the binary successor from a binary number with a leading 0 Papadimitriou, 1994: $M = \langle\{s, q\}, \{0, 1, \sqcup, \rhd\}, \delta, s\rangle$ with $\delta$ as follows:

| $s_i \in S$ | $\sigma \in \Sigma$ | $\delta(s_i, \sigma)$ |
|---|---|---|
| $s$ | 0 | $(s, 0, \rightarrow)$ |
| $s$ | 1 | $(s, 1, \rightarrow)$ |
| $s$ | $\sqcup$ | $(q, \sqcup, \leftarrow)$ |
| $s$ | $\rhd$ | $(s, \rhd, \rightarrow)$ |
| $q$ | 0 | $(h, 1, -)$ |
| $q$ | 1 | $(q, 0, \leftarrow)$ |
| $q$ | $\rhd$ | $(h, \rhd, \rightarrow)$ |

Now we can feed the Turing machine with some input (”” indicates the position of the tape head):

| step | state | tape |
|---|---|---|
| 0 | $s$ | $\underline{\rhd}110$ |
| 1 | $s$ | $\rhd\underline{1}10$ |
| 2 | $s$ | $\rhd1\underline{1}0$ |
| 3 | $s$ | $\rhd11\underline{0}$ |
| 4 | $s$ | $\rhd110\underline{\sqcup}$ |
| 5 | $q$ | $\rhd11\underline{0}\sqcup$ |
| 6 | $h$ | $\rhd11\underline{1}\sqcup$ |

| step | state | tape |
|---|---|---|
| 0 | $s$ | $\underline{\rhd}111$ |
| 1 | $s$ | $\rhd\underline{1}11$ |
| 2 | $s$ | $\rhd1\underline{1}1$ |
| 3 | $s$ | $\rhd11\underline{1}$ |
| 4 | $s$ | $\rhd111\underline{\sqcup}$ |
| 5 | $q$ | $\rhd11\underline{1}\sqcup$ |
| 6 | $q$ | $\rhd1\underline{1}0\sqcup$ |
| 7 | $q$ | $\rhd\underline{1}00\sqcup$ |
| 8 | $q$ | $\underline{\rhd}000\sqcup$ |
| 9 | $h$ | $\rhd\underline{0}00\sqcup$ |

So we now have an intuition how a Turing machine works, but what is still missing is a formal definition. To speak about how modifying the current configuration of a machine we first have to define what are the characteristics of such a configuration. Intuitively states of a Turing machine are characterized by the state, the string and the cursor position on the tape. This is captured by Definition 2.1.3.

**Definiton 2.1.3** (configuration)**.** *Let $M = \langle S, \Sigma, \delta, s\rangle$ be a fixed Turing machine. A configuration of this machine is a triple $(q, w, u)$, where $q \in S$ is the current state and $w, u \in \Sigma^*$ are strings. $u$ is the string to the left of the cursor, including the symbol under the cursor and $w$ is the string to the right of the cursor.*

As an example we notate the initial configuration $(s, \triangleright, x)$. Using the concept of configuration, Definition 2.1.4 formalizes computation steps.

**Definiton 2.1.4** (yields in one step)**.** *We say a configuration $(q, w, u)$ with $w = w_1 \dots w_n$ and $u = u_1 \dots u_m$ yields a configuration $(q', w', u')$ in one step iff one of the following conditions holds:*

1. $\delta(q, w_n) = (q', \rho, -)$, $w' = w_1 \dots w_{n-1}\ \rho$ *and* $u' = u$
2. $\delta(q, w_n) = (q', \rho, \rightarrow)$, $w' = w_1 \dots w_{n-1}\ \rho\ u_1$ *and* $u' = u_2 \dots u_m$
3. $\delta(q, w_n) = (q', \rho, \leftarrow)$, $w' = w_1 \dots w_{n-1}$ *and* $u' = \rho\ u_1 \dots u_m$

*We denote yields in on step by $(q, w, u) \overset{M}{\rightarrow} (q', w', u')$.*

We now use the transitive closure of yields in one step to define the more general yields relation.

**Definiton 2.1.5** (yields)**.** *We say one configuration $(q', w', u')$ yields a configuration $(q', w', u')$ in $k$-steps if there is a series $C_i$ of $k+1$ configurations such that $C_0 = (q, w, u)$, $C_{k+1} = (q', w', u')$ and for every $1 \leq i \leq k$ holds that $C_i \overset{M}{\rightarrow} C_{i+1}$. We denote this by $(q, w, u) \overset{M^k}{\rightarrow} (q', w', u')$.*
*We say a configuration $C$ yields to another configuration $C'$ if there exists a positive integer $k$ such that $C \overset{M^k}{\rightarrow} C'$ and denote this by $C \overset{M^*}{\rightarrow} C'$.*

*Example* 2.1.6. Back to our Turing machine $M$ for computing the binary successor from some input. If we start $M$ on $x$ we denote the computation like this:
for input $x = 110$:
$(s, \triangleright, 110) \overset{M}{\rightarrow} (s, \triangleright 1, 10) \overset{M}{\rightarrow} (s, \triangleright 11, 0) \overset{M}{\rightarrow} (s, \triangleright 110, \epsilon) \overset{M}{\rightarrow} (s, \triangleright 110 \sqcup, \epsilon) \overset{M}{\rightarrow} (q, \triangleright 110, \sqcup) \overset{M}{\rightarrow} (h, \triangleright 111, \sqcup)$
for input $x = 111$:
$(s, \triangleright, 111) \overset{M}{\rightarrow} (s, \triangleright 1, 11) \overset{M}{\rightarrow} (s, \triangleright 11, 1) \overset{M}{\rightarrow} (s, \triangleright 111, \epsilon) \overset{M}{\rightarrow} (s, \triangleright 111 \sqcup, \epsilon) \overset{M}{\rightarrow} (q, \triangleright 111, \sqcup) \overset{M}{\rightarrow} (q, \triangleright 11, 0 \sqcup) \overset{M}{\rightarrow}$
$(q, \triangleright 1, 00 \sqcup) \overset{M}{\rightarrow} (q, \triangleright, 000 \sqcup) \overset{M}{\rightarrow} (h, \triangleright 0, 00 \sqcup)$

Using the former definitions we can give a formal definition of a Turing machine's output in Definition 2.1.7.

**Definiton 2.1.7** (output)**.** *The Output of a Turing machine $M$ with input $x$ is defined as*

$$\mathrm{M}(x) = \begin{cases} \text{“}yes\text{”} & (s, \triangleright, x) \overset{M^*}{\rightarrow} (\text{“}yes\text{”}, u, w) \\ \text{“}no\text{”} & (s, \triangleright, x) \overset{M^*}{\rightarrow} (\text{“}no\text{”}, u, w) \\ y & (s, \triangleright, x) \overset{M^*}{\rightarrow} (h, \triangleright, y) \end{cases}$$

### Computable Functions

Given an input $x$, Turing machines can accept/reject it or compute some output. This leads us to the following definitions of recursive languages and functions.

**Definiton 2.1.8** (recursive language)**.** *A language $L$ is a set of strings $L \subset (\Sigma - \sqcup)^*$. We say a Turing $M$ decides the language $L$ iff for any $x \in (\Sigma - \sqcup)^*$ holds that $M(x) =$ "yes" if $x \in L$ and $M(x) =$ "no" otherwise.*
*We call a language $L$ a recursive/decidable language if there exists a Turing machine that decides $L$.*

We have to mention this notion of languages captures even more than just natural languages. A language may be the encodings of graphs, boolean circuits or logical formulas and the Turing machines decides for example, if the graph is connected, the output gate is true or the formula is satisfiable.

But there are more problems than just these decision problems. Many natural problems can't be solved by a yes/no answer, but by a number or a string encoding. This leads us to string functions and their computability.

**Definiton 2.1.9** (recursive function)**.** *A string function $f$ from $(\Sigma - \sqcup)^*$ to $\Sigma^*$ is called a recursive / computable function iff there exists a Turing machine $M$ with $M(x) = f(x)$ for all $x \in (\Sigma - \sqcup)^*$. We say $M$ computes $f$.*

As we mentioned for recursive languages the function $f$ is not necessarily a mathematical function. It may map encodings of graphs to the length of the shortest tour or even to the encoding of such a tour.

The Church-Turing thesis captures why the computability of Turing machines is so interesting for us.

**Theorem 2.1.10** (Church-Turing thesis)**.** *Every effectively calculable function is a recursive function.*

The Church-Turing thesis is not a provable statement, it is a common belief. This thesis is confirmed by the fact that many other reasonable computation models like $\lambda$-calculus, $\mu$-recursive functions and register machines leads to equivalent computability terms.

### k-String Machines

Sometimes the generalized machine model of k-String Turing machines is useful, for example it is necessary when distinguish the space for the input from space used for the computation. A k-String Turing machine is a deterministic Turing machine operating on $k$ tapes instead of only one.

**Definiton 2.1.11.** *A k-string Turing machine (TM) is a tuple*

$$M = \langle S, \Sigma, \delta, s \rangle$$

*with:*

- $S, \Sigma, s$ *defined as for ordinary deterministic Turing machines.*
- $\delta$ *a (partial) transition function:*

$$\delta : S \times \Sigma^k \rightarrow S \cup \{\text{“yes”}, \text{“no”}, h\} \times (\Sigma \times \{\rightarrow, \leftarrow, -\})^k$$

The configuration of the k-string machine is defined analogously to normal Turing machines. So a configuration is a $(2k+1)$-tuple $(q, w_1, u_1, \ldots w_k, u_k)$ where $q$ is the state and $w_i u_i$ the $i$th string and the $i$th cursor is on the last symbol of $w_i$.

The model of k-String Turing machines is a direct generalization of the simple Turing machine. If we set $k = 1$ we get a simple Turing machine. Therefore it suffices to define/ prove properties for k-String machines.

**Definiton 2.1.12** (time)**.** *Let $M$ be a k-String Turing machine and $(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \overset{M^t}{\rightarrow} (H, w_1, u_1, \ldots, w_k, u_k)$ holds for some input $x$ and halting state $H \in \{accept, reject, h\}$. The time required by $M$ on input $x$ is $t$.*
*If $M$ does not halt on $x$, $M(x) = \nearrow$, we define the time to be $\infty$.*

Theorem 2.1.13 says that the expressive power of k-String machines is not stronger than the expressive power of ordinary Turing machines. They are only a little bit faster.

**Theorem 2.1.13.** *Given a k-string Turing machine $M$, we can construct a Turing machine $M'$ with $M(x) = M'(x)$ for all inputs $x$. Let $n$ be the size of the input. If $M$ needs $O(f(n))$ steps for the computation on a arbitrary input we can construct $M'$ that works in $O(f(n)^2)$*

*Proof.* See Papadimitriou, 1994 Theorem 2.1. □

Beside time there are some other resources used by an algorithm, which we are interested in. These are the space required by the computation, and ,if analyzing parallel computing, the number of processors. We don't care about parallel computing here but space bounds plays an important role in this thesis.

If we measure the space needed by an algorithm it would be unfair to include the size of the input or the size of the output. In practice we may print the output symbols immediately and so have no reason to save it. Our machine model makes no difference between input, output and space used for the computation. As we want to define the space usage consider the former comments we have to improve the model of k-string Turing machines slightly.

**Definiton 2.1.14.** *A k-String Turing machine with input and output is an ordinary k-String Turing machine, with some restrictions to the function $\delta$:*

$$\delta(q, \sigma_1, \ldots, \sigma_k) = (p, \sigma_1, D_1, p_2, D_2, \ldots, p_k, D_k)$$

*with $D_k \neq \leftarrow$ and if $\sigma_1 = \sqcup$ then $D_1 = \leftarrow$*

If we assume the first tape as input tape and the last tape as output tape this definition implements that the machine does not modify the content on the input tape or move the input cursor out of the "input area" into the $\sqcup$-symbols. Further the restriction on $D_k$ prohibits the access to the output we have already written.

The next theorem guaranties that the input and output improvement does not change the time bounds of a machine significantly.

**Theorem 2.1.15.** *Let $M$ be a $k$-string Turing machine operating in some time bound $f(n)$, with $n$ the size of the input. We can find a $(k+2)$-string Turing machine $M'$ with input and output that operates in time $O(f(n))$.*

*Proof.* The machine $M'$ first copies the input on the second string. Then it simulates the machine $M$ on the tapes $2, \dots k+1$. When the simulation of $M$ halts $M'$ completes the computation by copying the output to the last string.
Copying the input is in linear time and therefore in $O(f(n))$. (assuming $f(n) > n$)
Simulating $M$ is in $O(f(n))$ by the precondition. The size of the output is bounded by the time of the computation and therefore copying the output is also in $O(f(n))$. $\square$

With this improved Turing machine we define the space used by a computation. In the following we denote the length of a string $x$ as $|x|$.

**Definiton 2.1.16** (space). *Let $M$ be a $k$-string Turing machine with input and output tape. The space of a computation on input $x$ $(s, \triangleright, x, \triangleright, \epsilon, \dots, \triangleright, \epsilon) \overset{M^*}{\rightarrow} (H, w_1, u_1, \dots, w_k, u_k)$, with $H$ a halting state, is defined as $\sum_{i=2}^{k-1} |w_i u_i|$.*

### 2.1.2 Nondeterministic Turing Machine

So far we looked on machines that seemed to be quite natural models of "computers". Beside these deterministic machines there are some other useful models which may seem unnatural and can't be mapped to a common programming language but are very useful in computational complexity theory.

In this section we generalize Turing machines by adding nondeterminism to the ordinary Turing machine. This leads us to the definition of nondeterministic Turing machines.

**Definiton 2.1.17.** *A nondeterministic Turing machine (NTM) is a tuple $M = (K, \Sigma, \delta, s)$*

$$N = \langle S, \Sigma, \delta, s \rangle$$

*with:*

- *$S$ a finite set of states*
- *$\Sigma$ the alphabet of $M$ - a finite set of symbols including the symbols $\sqcup, \triangleright$*

- *$\delta$ a (partial) transition relation:*

$$\delta \subseteq (S \times \Sigma) \times [S \cup \{\text{“yes”}, \text{“no”}, h\} \times \Sigma \times \{\rightarrow, \leftarrow, -\}]$$

- *$s \in S$ the initial state*

In a nondeterministic Turing machine there are in every step several possible actions the machine may choice. This is realized in the definition by replacing the function $\delta$ with a relation. This enforces an adaptation of “yields” for nondeterministic Turing machines.

**Definiton 2.1.18** (yields in one step)**.** *Let $N$ be a nondeterministic machine: We say a configuration $(q, w, u)$ with $w' = w_1 \dots w_n$ and $u = u_1 \dots u_m$ yields a configuration $(q', w', u')$ in one step ($(q, w, u) \overset{N}{\rightarrow} (q', w', u')$) iff one of the following conditions holds:*

1. *$(q, w_n, q', \rho, -) \in \delta$, $w' = w_1 \dots w_{n-1}\ \rho$ and $u' = u$*
2. *$(q, w_n, q', \rho, \rightarrow) \in \delta$, $w' = w_1 \dots w_{n-1}\ \rho\ u_1$ and $u' = u_2 \dots u_m$*
3. *$(q, w_n, q', \rho, \leftarrow) \in \delta$, $w' = w_1 \dots w_{n-1}$ and $u' = \rho\ u_1 \dots u_m$*

The terms “yields in k steps” and “yields” are defined as transitive closure to “yields in one step” in the same way as we did for deterministic machines.

We notice that every deterministic Turing machine is also a nondeterministic Turing machine as every function $\delta$ is also a relation. Furthermore the “yields” definitions for nondeterministic machines are compatible with those from the deterministic machines. The model of a nondeterministic Turing machine is a generalization of ordinary Turing machines.

So far, it is not clear what the output of such a machine would be. As there are many possible computations there may be different outputs and accepting states for the same input. We are mainly interested in decision problems, so we solve this inconsistency only for accepting languages.

**Definiton 2.1.19** (decide)**.** *Let $N$ be a nondeterministic machine and $L \subset (\Sigma - \sqcup)^*$ a language. $N$ decides $L$ if it holds for all $x \in \Sigma^*$ that $x \in L$ iff $(s, \triangleright, x) \overset{N^*}{\rightarrow} (\text{“yes”}, w, u)$*

So we accept an input $x$ if there exists an accepting computation and rejects if all possible computations reject. This is an asymmetry between “yes” and “no” instances.

**Theorem 2.1.20.** *Let $L$ be a language decided by a nondeterministic Turing Machine $N$ in time $f(n)$. There exists a 3-string deterministic Turing machine $M$ that decides $L$ in time $O(c^{f(n)})$, with $c > 0$ a constant depending of $N$.*

*Proof idea.* In each computation step, for each $(q, \sigma)$, there is only a finite number $d_{q,\sigma}$ of choices. The ”degree of nondeterminism” of N is the maximum value of these $d = \max\limits_{q,\sigma} d_{q,\sigma}$

Every computation of the machine $N$ is basically a sequence of nondeterministic choices. We can represent each choice as a integer in $\{0, 1, \dots, d-1\}$ and therefore each nondeterministic

computation as sequence of integers.
The idea for simulating $N$ is to consider all sequences of choices, in order of increasing length, and simulate $N$ on them. If we find a sequence that $N$ accepts the input $x$ then $M$ halts with "yes". In this case we found a computation of $N$ that accepts $x$ and so accepting the input is the right thing for $M$.

But its not so easy to verify if $M$ should reject the input. It not suffices to find a choice that rejects the input we have to verify that all choice sequences lead to reject. What we have to do is checking if we already have tested all relevant choice sequences. If $M$ has simulated all choice sequences of a specific length and all of them halt with "no" then $M$ returns "no".
How to handle all this computations:
In every simulation of $N$ we have to protect our input for later simulations and additionally we need some place to save the current sequence of choices.
To partition these things, we use tape 1 as a input tape, save the sequence of choices on tape 2 and make our computation on tape 3 of $M$. □

If a problem or language can be decided by a nondeterministic Turing machine, there is also a deterministic Turing machine deciding the same problem. So the non deterministic machine gives no additional power when studying computability of problems.

In the former theorem there is a exponential time blow up when simulating the nondeterministic Turing machine with a deterministic Turing machine so this makes a significant difference when comparing the necessary running time of problems. We should mention here that we don't know if this exponential blow up is necessary or not, but it is a common belief. This question has a strong connection to one of the most important open problems in theoretical computer science, the problem $\mathsf{P} \neq \mathsf{NP}$.

## 2.2 Basic Concepts of Complexity Theory

We have already used Turing machines to distinguish decidable languages / computable functions from languages / functions that we can't solve algorithmically. Computational complexity theory is not only interested in the solvability of problems it is further interested in the complexity of deciding a problem or computing a function. We start this section with discussing what the definition of problem in computational complexity theory is. Further we present the concepts of complexity classes, problem reductions and completeness which are used to measure the computational complexity of problems and subdivide the set of decidable problems according to their different complexity.

### 2.2.1 Problems

Computer scientists often analyze algorithms and their run time and space requirements relating to the size of the input. This is good for comparing algorithms solving the same problem or showing that a problem is efficiently solvable. But this can't be used to prove that no efficient algorithm exists and we may waste time searching for one. So complexity theory focuses on the computational complexity of problems and not on the complexity of algorithms.

So first we have to specify what our understanding of a problem is.

**Definiton 2.2.1** (problem)**.** *In complexity theory a problem specification consist of two parts. A criterion that defines an infinite set of possible instances and a question on these instances.*

In the most cases we deal with decision problems. A decision problem is a problem containing a question that can be answered either by yes or no. To make this definition more concrete we give graph reachability as ansimple example for a decision problem:

**REACHABILITY**

***Given:*** A graph $G = (V, E)$ and two nodes $v_1, v_2 \in V$

***Problem:*** Is there a path from $v_1$ to $v_2$?

Here the infinite set of instances is the set of graphs with two marked nodes. We can't feed this problem direct to a Turing machine. We have to find a reasonable encoding of the problem in the machines alphabet $(\Sigma - \sqcup)$. So the decision problem maps to a language acceptance problem of the encodings.

### 2.2.2 Complexity Classes

A complexity class basically is set of problems having the same computational complexity. To define such a class we usually first fix a computation model, for example k-string Turing machines or nondeterministic machines. Then we chose the mode of computation, the way to determine if the machine accepts an input. The different modes we already defined are the deterministic (Definition 2.1.8) and the nondeterministic mode (Definition 2.1.19) of computation. We define one more nondeterministic and the alternating mode later when discussing alternating Turing machines. In the third step we chose the resource we want to bound, in the most cases time or space, and of course a bound $f(n)$ for the resource.

The complexity class is the set of languages / problems decided by such a computation model operating in the specified mode and for any input $x$ the computation does not exceed the resource bound $f(n)$.

To avoid technical problems and counterintuitive effects we restrict the functions $f$ used for resource bounds to the class of proper complexity functions. Intuitively we assume that a larger input would need more computation resources therefore it is natural to assume a function $f$ to be increasing or at least non decreasing. Further we claim the function $f$ to be efficiently computable.

**Definiton 2.2.2** (proper complexity functions)**.** *We call a function $f$ from the nonnegative integers to the nonnegative integers a proper complexity function if $f$ is non decreasing and the following conditions hold. There exists a k-string Turing machine $M_f$ mapping any input $x$ to $\sqcap^{f(|x|)}$ in time $O(|x| + f(|x|))$ and space $O(f(|x|))$.*

*The Turing machine $M_f$ operates in linear time according to the size of the input and the output and needs linear space according to the size of the output. (encoding the value of $f(n)$ in unary)*

In the following we always assume our resource bound functions $f$ to be proper complexity functions.

### Time Complexity

The first important resource for computation is time.To get complexity classes distinguishing problems with respect to the time required for computation, we first fix our computation model to k-string Turing machines with the deterministic mode of computation. In the following $n$ denotes the string length of the input.

**Definiton 2.2.3** (TIME)**.** *Let $L \subset (\Sigma - \{\sqcup\})^*$ be a language. If $L$ is decided by a deterministic (k-string) Turing machine working in time $O(f(n))$ we write $L \in \mathsf{TIME}(f(n))$. So $\mathsf{TIME}(f(n))$ is the set of languages that can be decided by deterministic Turing machines within time bound $f(n)$.*

So this is a kind of asymptotic worst case complexity. A problem is in the complexity class $\mathsf{TIME}(f(n))$ only if every problem instance is decided in $f(n)$. So as soon there is one instance not decided in $f(n)$ the problem is not in $\mathsf{TIME}(f(n))$.
We define time complexity classes for nondeterministic Turing machines analogously:

**Definiton 2.2.4** (NTIME)**.** *Let $L \subset (\Sigma - \{\sqcup\})^*$ be a language. If $L$ is decided by a nondeterministic Turing machine working in time $O(f(n))$ we write $L \in \mathsf{NTIME}(f(n))$. So $\mathsf{NTIME}(f(n))$ is the set of languages that can be decided by nondeterministic Turing machines within time bound $f(n)$.*

### Space Complexity

We have already mentioned that there are other important resources beside the time used by a machine. Here we introduce complexity classes measuring the space used by a deterministic Turing machine in Definition 2.2.5 and by a nondeterministic Turing machine in Definition 2.2.6.

**Definiton 2.2.5** (SPACE)**.** *Let $L \subset (\Sigma - \{\sqcup\})^*$ be a language. If $L$ is decided by a deterministic Turing machine with input working in space $f(n)$ we write $L \in \mathsf{SPACE}(f(n))$. So $\mathsf{SPACE}(f(n))$ is the set of languages that can be decided by deterministic Turing machines within space bound $f(n)$.*

**Definiton 2.2.6** (NSPACE)**.** *Let $L \subset (\Sigma - \{\sqcup\})^*$ be a language. If $L$ is decided by a nondeterministic Turing machine with input working in space $f(n)$ we write $L \in \mathsf{NSPACE}(f(n))$. So $\mathsf{NSPACE}(f(n))$ is the set of languages that can be decided by nondeterministic Turing machines within space bound $f(n)$.*

### 2.2.3 Problem Reductions

In computational complexity theory we want to compare the difficulty of problems or classes. So we need a concept which tells us that a problem is harder or at least as hard as another. The basic idea is to reduce problems to other problems. A reduction is a function $R$ that transforms the instances $a_i$ of a problem $A$ to instances of another problem $B$ such that $a_i$ is true iff $R(a_i)$ is true.

If problem $A$ can be reduced to another problem $B$ then $A$ can't be harder than $B$ together with the effort for the reduction. This is because problem $A$ can be solved by applying the reduction and an algorithm for problem $B$. Using this argumentation we have to care about the resources used by the reduction. If we allow arbitrary functions $R$ for reductions we can put the complexity of a very hard problem in the function $R$ and reduce it to a much simpler problem. So we have to restrict reductions to functions that are computable in the restrictions of the lowest complexity classes we are interested in. For our purposes log-space reduction suffices:

**Definiton 2.2.7** ($\log n$ reduction)**.** *A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ computable by a deterministic Turing machine in space $O(\log n)$ and it holds that $x \in L_1$ iff $R(x) \in L_2$. The function $R$ is called a reduction from $L_1$ to $L_2$.*

Sometimes reductions are defined as polynomial time computable functions instead of the $\log n$ space bound. We mention here that our definition of reduction is more restrictive, all $\log n$ reductions are polynomial time algorithms.

**Theorem 2.2.8.** *Let $R$ be a $\log n$-reduction computed by a Turing machine $M$. $M$ halts after a polynomial number of steps on each input $x$.*

*Proof.* There are $O(n \cdot c^{\log n})$ possible configurations for $M$ and in deterministic machines a configuration can't repeat in a computation (assuming the machine halts). □

**Theorem 2.2.9.** *Let $f$ be an reduction from $L_1$ to $L_2$ and $g$ an reduction from $L_2$ to $L_3$ then $f \circ g$ is a reduction from $L_1$ to $L_3$*

*Proof.* See Papadimitriou, 1994 Proposition 8.2 □

### 2.2.4 Completeness

We defined our complexity classes as set of languages / problems decidable in some time and space bounds. This is just an upper bound for complexity and so there are problems with different hardness in the same complexity class. To get lower bounds for complexity we define the hardness of a problem.

**Definiton 2.2.10** (hardness)**.** *Given a complexity class* $\mathsf{C}$ *and a language $L$.*
*We say $L$ is* $\mathsf{C}$*-hard if any $L^{'} \in C$ can be reduced to $L$*

We use the hardness term to define representative problems for a complexity class:

**Definiton 2.2.11** (completeness)**.** *Given a complexity class* $\mathsf{C}$ *and a language* $L \in \mathsf{C}$.
*We call* $L$ $\mathsf{C}$*-complete if any language* $L^{'} \in C$ *can be reduced to* $L$.

So a problem $P$ is complete for a complexity class $\mathsf{C}$ if $P$ is a member of the class $\mathsf{C}$ and $P$ is $\mathsf{C}$-hard.

**Definiton 2.2.12.** *A complexity class* $\mathsf{C}$ *is closed under reductions when the following property holds. If a language* $L$ *is reducible to a language* $L^{'} \in \mathsf{C}$ *then* $L \in \mathsf{C}$

We are only interested in complexity classes that are closed under reductions. Especially all complexity classes we define in the next section are closed under reductions.
The next theorem gives a strong connection between complexity classes and their complete problems:

**Theorem 2.2.13.** *Let* $\mathsf{C}, \mathsf{C}^{'}$ *be complexity classes closed under reductions. If there exists a language* $L$ *which is complete for both classes* $\mathsf{C}$ *and* $\mathsf{C}^{'}$ *then* $\mathsf{C} = \mathsf{C}^{'}$.

*Proof.* As this is a symmetric problem it suffices to show that $\mathsf{C} \subseteq \mathsf{C}^{'}$. The other inclusion follows by interchanging $\mathsf{C}$ and $\mathsf{C}^{'}$ in the proof.
Every language $L_C \in \mathsf{C}$ reduces to $L \in \mathsf{C}^{'}$, as $L$ is hard for the class $\mathsf{C}$. Further $\mathsf{C}^{'}$ is closed under reductions and therefore $L_C \in \mathsf{C}^{'}$. So it holds that $\mathsf{C} \subseteq \mathsf{C}^{'}$. □

## 2.3 Important Complexity classes

In this section we first give an overview of the important complexity classes based on deterministic and nondeterministic Turing machines. Then we present the few known relations between these complexity classes.

### 2.3.1 Deterministic Complexity Classes

First we define the classes based on the deterministic Turing machines. Deterministic Turing machines are believed to be realistic models for computations. Therefore we assume there is a good correspondence between the resources the computation needs on a Turing machine and the resources the computation needs on a real computer and therefore there is a good understanding of the complexity these classes provide.

To define the first complexity classes we choose bounds for the computation time. As every program should at least read the whole input the time bound must be at least a linear polynomial. So the lowest time complexity class closed under reductions is given by polynomial bounds:

**Definiton 2.3.1** ($\mathsf{P}$)**.** *The class* $\mathsf{P}$ *(polynomial time) is the class of problems decided by a deterministic Turing machine in polynomial time.*

$$\mathsf{P} = \mathsf{TIME}(n^k) = \bigcup_{j>0} \mathsf{TIME}(n^j)$$

Now we have the class P for the efficiently solvable problems. But there are many hard problems solved by algorithms requiring exponential time in relation to the input. This leads us to the following definition.

**Definiton 2.3.2** (EXPTIME). *The class* EXPTIME *(exponential time) is the class of problems decided by a deterministic Turing machine in exponential time.*

$$\mathsf{EXPTIME} = \mathsf{TIME}(2^{n^k}) = \bigcup_{j>0} \mathsf{TIME}(2^{n^j})$$

After defining the classes P and EXPTIME we can specify space bounded complexity classes in a similar way. The assumption that space is a more restricted resource than time is a motivation for using less increasing functions.

**Definiton 2.3.3** (L). *The class* L *(deterministic logarithmic space) is the class of problems decided by a deterministic Turing machine in logarithmic space.*

$$\mathsf{L} = \mathsf{SPACE}(\log n)$$

So logarithmic space seems to be a very strict restriction, for example we can't copy great parts of the inputs. But there are some important things that can be done in logarithmic space. First we can use variables which not depend on the input size. Second we consider that encoding numbers needs logarithmic size to their value and therefore we can use counters with values polynomial to the input size. Furthermore we can implement cursors to input symbols, using counters.
A complexity class with less restrictive space bounds is the class PSPACE.

**Definiton 2.3.4** (PSPACE). *The class* PSPACE *(deterministic polynomial space) is the class of problems decided by a deterministic Turing machine in polynomial space.*

$$\mathsf{PSPACE} = \mathsf{SPACE}(n^k) = \bigcup_{j>0} \mathsf{SPACE}(n^j)$$

### 2.3.2 Nondeterministic Complexity Classes

Using nondeterministic Turing machines with the ordinary nondeterministic computation mode we define nondeterministic complexity classes in the same way as for deterministic ones.

**Definiton 2.3.5** (NP). *The class* NP *(nondeterministic polynomial time) is the class of problems decided by a nondeterministic Turing machine in polynomial time.*

$$\mathsf{NP} = \mathsf{NTIME}(n^k) = \bigcup_{j>0} \mathsf{NTIME}(n^j)$$

It is believed, but not shown yet, that nondeterministic time complexity is not closed under complementation. This is the asymmetry between “yes” and “no” instances we mentioned in Section 2.1.2. So it makes sense to define the complementary class of NP:

**Definiton 2.3.6** (co-NP)**.** *The class* co-NP *is the class of languages $L$, with the complement $\bar{L}$ decided by a nondeterministic Turing machine in polynomial Time. When $L$ is a language over the alphabet $\Sigma$ the complement $\bar{L}$ is defined as $\bar{L} = \Sigma^* \setminus L$*

**Definiton 2.3.7** (NEXPTIME)**.** *The class* NEXPTIME *(nondeterministic exponential time) is the class of problems decided by a nondeterministic Turing machine in exponential time.*

$$\mathsf{NEXPTIME} = \mathsf{NTIME}(2^{n^k}) = \bigcup_{j>0} \mathsf{NTIME}(2^{n^j})$$

**Definiton 2.3.8** (NL)**.** *The class* NL *(nondeterministic logarithmic space) is the class of problems decided by a nondeterministic Turing machine in logarithmic space.*

$$\mathsf{NL} = \mathsf{NSPACE}(\log n)$$

**Definiton 2.3.9** (NPSPACE)**.** *The class* NPSPACE *(nondeterministic polynomial space) is the class of problems decided by a nondeterministic Turing machine in polynomial space.*

$$\mathsf{NPSPACE} = \mathsf{NSPACE}(n^k) = \bigcup_{j>0} \mathsf{NSPACE}(n^j)$$

### 2.3.3 Relations

Now we have a lot complexity classes but what is still missing is an ordering in the set of complexity classes. So what are ”good” complexity classes and what are classes for worse problems. Here we present the known relations between our complexity classes.

**Theorem 2.3.10.** *Let $f(n)$ be a proper complexity function, then the following relations hold:*

- $\mathsf{TIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$ *and* $\mathsf{SPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$
- $\mathsf{NTIME}(f(n)) \subseteq \mathsf{SPACE}(f(n))$
- $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(c^{\log n + f(n)})$

*Proof ideas.* a) Follows from the fact that every deterministic Turing machine is also a nondeterministic Turing machine.
b) Using the construction of a deterministic machine simulating the nondeterministic machine from Theorem 2.1.20. The sequence of choices can be saved in $O(f(n))$ and a time $f(n)$ machine can write at most $f(n)$ symbols.
c) W.l.o.g we assume that the nondeterministic machine $N$ has two tapes, an input and a work tape. Now we use the fact that the number of possible configurations is bounded by the space

used in the computation path. When counting the configurations there are $s = |S|$ possible states and $\sigma = |\Sigma|$ possible symbols for every tape cell. Further we have to mention the different cursor positions. There are $n + 1$ possible positions for the input cursor and $f(n)$ for the cursor on the work tape.
So we have $s \cdot \sigma^{f(n)} \cdot n \cdot f(n)$ as an upper bound. We can find a new constant $k$ such that the upper bound for the number of configurations simplifies to $k^{\log n + f(n)}$.
The deterministic machine $M$ just generates a graph $G = (V, E)$ with all possible configurations as nodes and edges as following. For two configurations $C_1, C_2$ the edge $(C_1, C_2) \in E$ iff $C_1 \overset{N}{\to} C_2$. To check if $N$ accepts the input, $M$ starts a reachability algorithm starting at the initial configuration of $N$ and searching for accepting configurations. $\square$

The next theorem gives us the reason to skip the complementary classes of nondeterministic space classes.

**Theorem 2.3.11.** *Immerman, 1988*

$$\mathsf{NSPACE}(f(n)) = \mathsf{co}{-}\mathsf{NSPACE}(f(n)) \qquad , \qquad f(n) \geq \log n$$

**Theorem 2.3.12** (Savitch's theorem)**.** *Savitch, 1970*

$$\mathsf{PSPACE} = \mathsf{NPSPACE}$$

In the next theorem we summarize the results in a hierarchy of our complexity classes.

**Theorem 2.3.13** (Relations)**.**

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$$

But not any of these relations is know to be proper. There are only a few results that complexity classes are proper included in another class, so it is proofed that they are not equal.

**Theorem 2.3.14.**

$$\mathsf{NL} \subset \mathsf{PSPACE}$$

$$\mathsf{P} \subset \mathsf{EXPTIME}$$

Further complexity classes and relations between them can be found in Aaronson et al., 2008.

## 2.4 Alternating Turing Machine

In this section we get in contact with alternation as a concept of computation. To introduce alternating Turing machines it is useful to have an alternative look at the already known concept of nondeterministic machines. We associate an acceptance attribute to each configuration which is computed recursively by its successors. With a successor of a configuration $(q, w, u)$ we mean

a configuration that is yielded in one step from $(q, w, u)$.
Now we give the definition how to compute the acceptance attribute of a configuration:

**Definiton 2.4.1** (leads to acceptance). *A configuration of a nondeterministic Turing machine* leads to acceptance *iff it is a final accepting configuration (the configuration includes the "yes" state) or at least one of its successors leads to acceptance.*
*A nondeterministic Turing machine accepts an input iff the initial configuration is an accepting configuration.*

We mention that this leading-to-acceptance notion is equivalent to the original yields notion. A nondeterministic Turing machine $N$ accepts an input $x$ according to leading to acceptance notion if and only if $N$ accepts $x$ according to the yields notion. With this new notion we can interpret the leading to acceptance value of a configuration just as an OR of its successors. As we now have machines computing the accepting configurations with an OR functionality it is natural to ask what happens if we use ANDs to combine the results of the successors.

Let $L$ be a language decided by a nondeterministic Turing machine $N$. Then there exists a nondeterministic machine $M$ deciding the complement of $L$ using an AND acceptance rule. AND acceptance rule here means that a configuration is accepted only if all successors are accepted. We can easily construct such a machine $M$ by interchanging the "yes","no" states in $N$.

Using this we give another characterization of the class co-NP using the AND computation mode. The class co-NP is the class of languages decided by a nondeterministic Turing with the AND computation mode machine in polynomial time.

The concept of alternating Turing machines, introduced by Chandra et al., 1981, allows both types of configurations. So in an alternating machine there are universal states, which accept a configuration only if all successors are accepted, and existential states, which accept a configuration if there is at least one accepted successor.
In the following we give a formal definition of such an alternating Turing machine:

**Definiton 2.4.2.** *An alternating Turing machine (ATM) with $k$ strings is a tuple*

$$A = (S, \Sigma, \delta, s, g)$$

*with:*

- *$S$ a finite set of states - not including the states "yes" and "no"*
- *$\Sigma$ the alphabet of $M$ - a finite set of symbols including the symbols $\sqcup, \triangleright$*
- *$\delta$ a (partial) transition relation:*

$$\delta \subseteq (S \times \Sigma \times \Sigma^k) \times \left[ S \cup \{\text{"yes"}, \text{"no"}, h\} \times \Sigma^k \times \{\rightarrow, \leftarrow, -\}^{k+1} \right]$$

- *$s \in S$ the initial state*

- *$g$ a function that maps states to boolean functions: $S \to \{\wedge, \vee, accept, reject\}$*

So an alternating machine basically is a nondeterministic Turing machine with some labels on the states. We call a state labeled with $\wedge$ universal state and a state labeled with $\vee$ existential state. Furthermore we assume that only the "yes" state is labeled with *accept* and only the "no" state is labeled with *reject*.

We use the labeling $g$ to define the acceptance-rule for non-deterministic Turing machines in the following way:

**Definiton 2.4.3** (leading to acceptance)**.** *We define the leading to acceptance value of an alternating Turing machine recursively:*

- *A configuration with state "yes" leads to acceptance; a configuration with state "no" does not lead to acceptance.*

- *A configuration with a universal state leads to acceptance if and only if all successor configurations lead to acceptance*

- *A configuration with an existential state leads to acceptance if at least one successor leads to acceptance.*

*An alternating Turing machine accepts an input iff the related initial configuration leads to acceptance. Sometime we say a configuration leads to rejection instead of the configuration does not lead to acceptance.*

An alternating Turing machine that uses only existential states and no universal states is equivalent to an ordinary nondeterministic Turing machine. So an alternating Turing machine is a generalization of nondeterministic Turing machines. As we have already mentioned we can use universal state to construct nondeterministic machines deciding the co-problems of ordinary nondeterministic problems. An alternating Turing machine with no existential state is equivalent to a nondeterministic machine deciding the co-problems.

The original definition of alternating machines in Chandra et al., 1981 also includes negating states. A negating state is a state with exactly one successor and a negating state accepts/rejects if its successor rejects/accepts. They further show that for every alternating Turing machine with negating states there is an equivalent alternating Turing machine without negating states working in the same time and space bounds. Therefore negation gives us no additional power and so we can restrict our alternating Turing machines to universal and existential states.

Further having an alternating Turing machine $A$ solving a problem, there exists an alternating Turing machine working in the same resource bounds and deciding the complementary problem. To prove this, we just add a negating state $n$ as initial state to $A$ and extend $\delta$ such that the old initial configuration is the successor of the new one. Now we simple use the above result to eliminate negation. Therefore all alternating complexity classes are closed under complement.

When dealing with computations of alternating machines the concepts of computation paths and computation trees are often useful.

**Definiton 2.4.4** (computation path). *A computation path of an (alternating) Turing machine $M$ on input $x$ is a sequence of configurations $c_i$ with:*

1. *$c_0$ the initial configuration*
2. *$c_0 \overset{M}{\rightarrow} c_1 \overset{M}{\rightarrow} \dots \overset{M}{\rightarrow} c_n$*

**Definiton 2.4.5** (computation graph). *The computation graph of an (alternating) Turing machine $M$ on input $x$ is the smallest directed graph $G = (V, E)$ fulfilling the following conditions:*

1. *The initial configuration $c_{start}$ is the root of the tree and therefore $c_{start} \in V$*
2. *if $c_1 \in V$ and $c_1 \overset{M}{\rightarrow} c_2$ then $c_2 \in V$ and $(c_1, c_2) \in E$*

We should mention here that a computation path is not necessarily a path in the graph theoretic sense, because it may contain a configuration twice. For the same reason the computation graph in general is no graph theoretic tree (a cycle free graph). Modifying rule 2 such that for every edge there is a new node for $c_2$ even if this configuration is already in the graph leads to the definition of a computation tree.

**Definiton 2.4.6** (computation tree). *The computation tree of an (alternating) Turing machine $M$ on input $x$ is the smallest directed graph $G = (V, E)$ fulfilling the following conditions:*

1. *Every $v \in V$ has the form $v = (id, C)$ where $C$ is configuration*
2. *For the initial configuration $c_{start}$ there is the root node of the tree $(0, c_{start}) \in V$*
3. *For each $(id, c_i) \in V$ and $c_i \overset{M}{\rightarrow} c_j$ it holds that there is a $(id', c_j) \in V$ such that $(id, c_i)$ is the only predecessor.*

Figures 2.1 and 2.3 illustrate the difference between computation graphs and computation trees.

*Example* 2.4.7 (An alternating Turing machine). The following alternating Turing machine A tests if a string $x \in \{a, b\}^*$ fulfills the following condition. For each cell containing an $a$ either the left or right neighbor contains a $b$ and for each $b$ both neighbors cells contains an $a$.
Let $A = (S, \Sigma, \delta, s, g)$ be a 1-tape machine with $\delta$ defined as follows:

| $s_i \in S$ | $\sigma \in \Sigma$ | $\delta(s_i, \sigma)$ |
|---|---|---|
| $s$ | $\triangleright$ | $(r, \triangleright, \rightarrow)$ |
| $r$ | $a$ | $(r, a, \rightarrow)$ |
| $r$ | $a$ | $(r_a, a, -)$ |
| $r$ | $b$ | $(r, b, \rightarrow)$ |
| $r$ | $b$ | $(r_b, b, -)$ |
| $r$ | $\sqcup$ | $(\text{“yes”}, \sqcup, -)$ |
| $r_a$ | $a$ | $(c_b, a, \leftarrow)$ |
| $r_a$ | $a$ | $(c_b, a, \rightarrow)$ |
| $r_b$ | $b$ | $(c_a, b, \leftarrow)$ |

| $s_i \in S$ | $\sigma \in \Sigma$ | $\delta(s_i, \sigma)$ |
|---|---|---|
| $r_b$ | $b$ | $(c_a, b, \rightarrow)$ |
| $c_a$ | $a$ | $(\text{“yes”}, a, -)$ |
| $c_a$ | $b$ | $(\text{“no”}, b, -)$ |
| $c_a$ | $\triangleright$ | $(\text{“no”}, \triangleright, -)$ |
| $c_a$ | $\sqcup$ | $(\text{“no”}, \sqcup, -)$ |
| $c_b$ | $a$ | $(\text{“no”}, a, -)$ |
| $c_b$ | $b$ | $(\text{“yes”}, b, -)$ |
| $c_b$ | $\triangleright$ | $(\text{“no”}, \triangleright, -)$ |
| $c_b$ | $\sqcup$ | $(\text{“no”}, \sqcup, -)$ |

We define the function $g$ only on nondeterministic and halting states. For states with only one successor both quantifiers evaluate to an accept if and and only if the successor evaluates to an accept. So it has no effect to the acceptance value which of the labels $\wedge$ and $\vee$ we assign to the remaining states.

| $s_i \in S$ | $g(s_i)$ |
|---|---|
| $r$ | $\wedge$ |
| $r_a$ | $\vee$ |
| $r_b$ | $\wedge$ |
| “yes” | *accept* |
| “no” | *reject* |

Figure 2.1 illustrates how this machine works on input $aba$ and Figure 2.2 shows how the acceptance is computed.

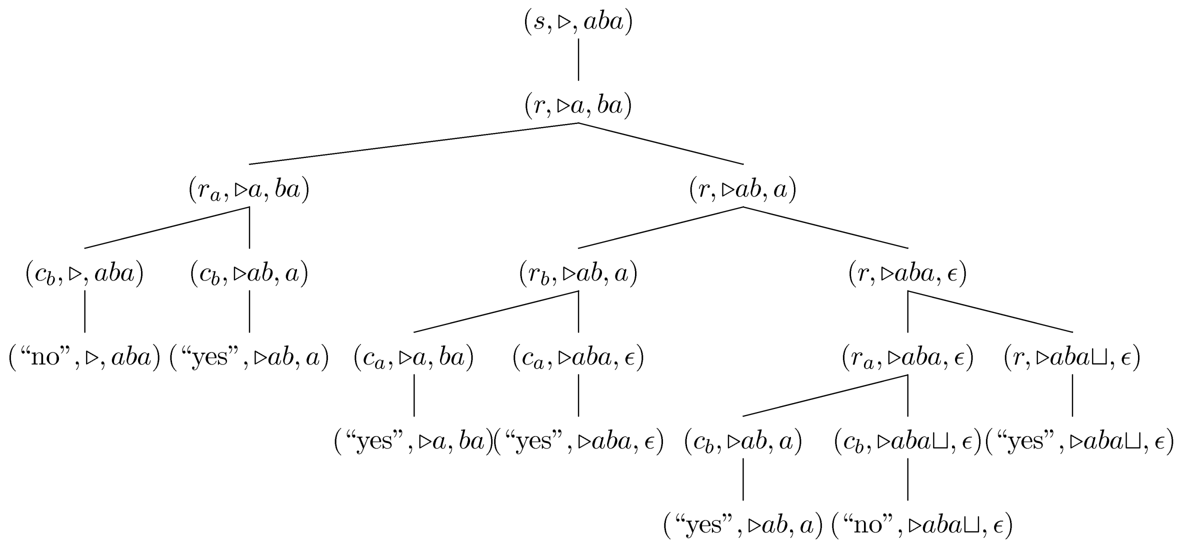

$(s, \triangleright, aba)$
$(r, \triangleright a, ba)$
$(r_a, \triangleright a, ba)$
$(r, \triangleright ab, a)$
$(c_b, \triangleright, aba)$
$(c_b, \triangleright ab, a)$
$(r_b, \triangleright ab, a)$
$(r, \triangleright aba, \epsilon)$
$(\text{“no”}, \triangleright, aba)$
$(\text{“yes”}, \triangleright ab, a)$
$(c_a, \triangleright a, ba)$
$(c_a, \triangleright aba, \epsilon)$
$(r_a, \triangleright aba, \epsilon)$
$(r, \triangleright aba\sqcup, \epsilon)$
$(\text{“yes”}, \triangleright a, ba)$
$(\text{“yes”}, \triangleright aba, \epsilon)$
$(c_b, \triangleright ab, a)$
$(c_b, \triangleright aba\sqcup, \epsilon)$
$(\text{“yes”}, \triangleright aba\sqcup, \epsilon)$
$(\text{“yes”}, \triangleright ab, a)$
$(\text{“no”}, \triangleright aba\sqcup, \epsilon)$

**Figure 2.1:** Computation tree of the computation with input $aba$ on the alternating Turing machine from Example 2.4.7.

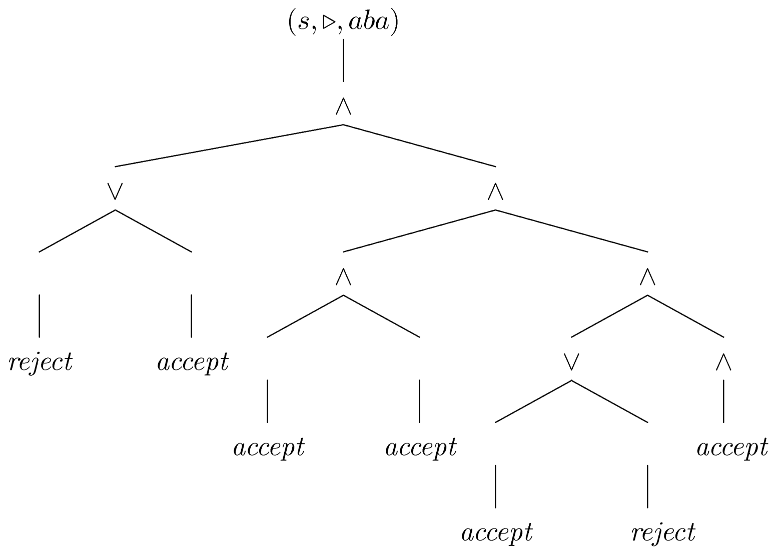
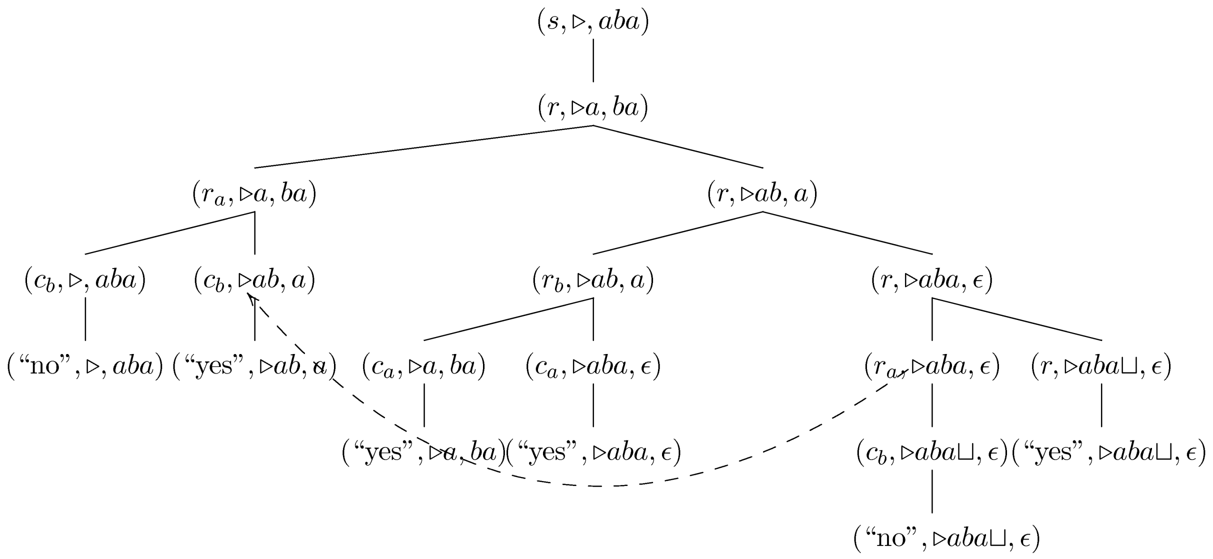


**Figure 2.2:** The way how the acceptance is computed on the computation tree in Figure2.1 from Example 2.4.7.



$(s, \triangleright, aba)$
$(r, \triangleright a, ba)$
$(r_a, \triangleright a, ba)$
$(r, \triangleright ab, a)$
$(c_b, \triangleright, aba)$
$(c_b, \triangleright ab, a)$
$(r_b, \triangleright ab, a)$
$(r, \triangleright aba, \epsilon)$
$(\text{“no”}, \triangleright, aba)$
$(\text{“yes”}, \triangleright ab, a)$
$(c_a, \triangleright a, ba)$
$(c_a, \triangleright aba, \epsilon)$
$(r_a, \triangleright aba, \epsilon)$
$(r, \triangleright aba\sqcup, \epsilon)$
$(\text{“yes”}, \triangleright a, ba)$
$(\text{“yes”}, \triangleright aba, \epsilon)$
$(c_b, \triangleright aba\sqcup, \epsilon)$
$(\text{“yes”}, \triangleright aba\sqcup, \epsilon)$
$(\text{“no”}, \triangleright aba\sqcup, \epsilon)$

**Figure 2.3:** Computation graph of the computation with input *aba* on the alternating Turing machine from Example 2.4.7.

### 2.4.1 Short-Circuit Evaluation

To compute the acceptance value of a configuration we so far compute the acceptance value of all successors and then combine them with a logical AND or logical OR. First this method seems inefficient for practical computation, which is no problem when dealing with asymptotic worst case complexity, but furthermore it fails as soon as one successor configuration leads to a non halting computation and can't be evaluated.

In the case of an existential configuration with an accepting successor we won't care about the other successors and similar holds for universal configurations. If there is one rejecting successor we won't care about the other successors. So we can decide the acceptance of quantified configurations with non halting successors as long we have a witness for the leading to acceptance value.

To give a formal definition of this concept we define functions that map every configuration to 1 (leading to acceptance), 0 (leading to rejection) or $\perp$ (don't care). So we have to extend the logic operators $\vee, \wedge$ to the new domain:

| $\wedge$ | 1 | $\perp$ | 0 |
|---|---|---|---|
| 1 | 1 | $\perp$ | 0 |
| $\perp$ | $\perp$ | $\perp$ | 0 |
| 0 | 0 | 0 | 0 |

| $\vee$ | 1 | $\perp$ | 0 |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| $\perp$ | 1 | $\perp$ | $\perp$ |
| 0 | 1 | $\perp$ | 0 |

Now we give the definition of a labeling that associates leading to acceptance values to the configurations.

**Definiton 2.4.8** (labeling of configurations)**.** *Let $A$ be an alternating Turing machine and $C_A$ the set of configurations from the computation of $x$. A labeling of configurations is a function* $\mathrm{l} : M \to \{0, \perp, 1\},\ M \subseteq C_A$.

To fulfill the definition of leading to acceptance a labeling l that decides the output of our Turing machine must label the initial configuration to 1 or 0 and satisfy the following condition.

$$\mathrm{l}(\alpha) = \begin{cases} \bigwedge_{\alpha \xrightarrow{A} \beta} \mathrm{l}(\beta) & \text{if } \alpha \text{ is universal} \\ \bigvee_{\alpha \xrightarrow{A} \beta} \mathrm{l}(\beta) & \text{if } \alpha \text{ existential} \\ 1 & \text{if } \alpha \text{ is accepting} \\ 0 & \text{if } \alpha \text{ is rejecting} \end{cases} \tag{2.1}$$

We assume that successors $\alpha$ not in the range of the labeling to be labeled with $\perp$ for this equation. ($\alpha \notin M \to l(\alpha) = \perp$).

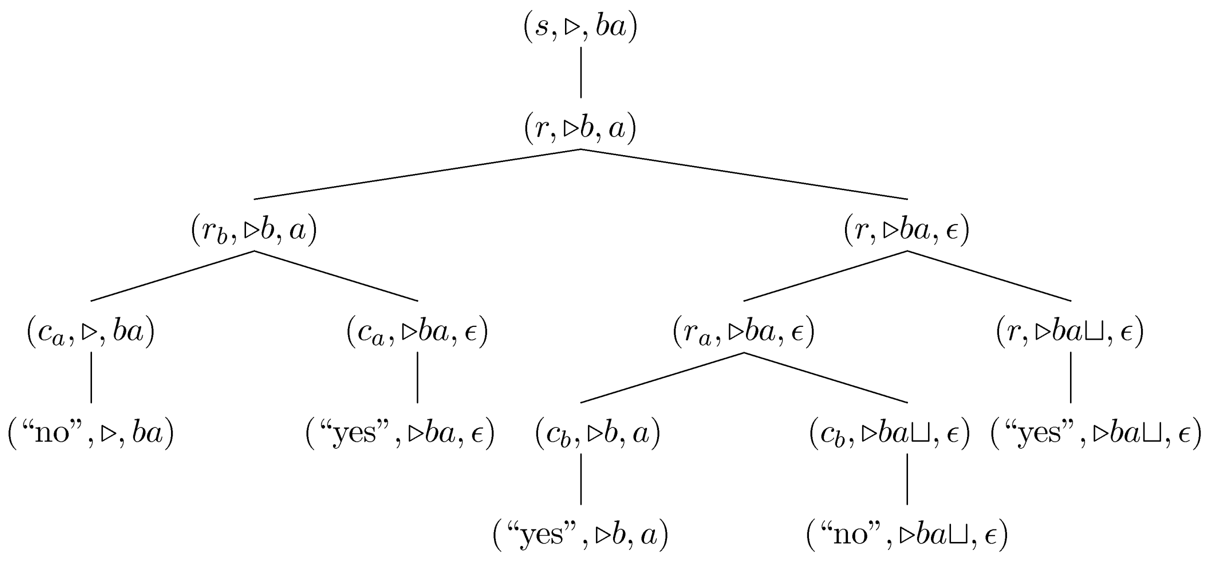
In Figure 2.4 we see an example illustrating the benefit of short-circuit evaluation compared to the naive evaluation.

### 2.4.2 Alternating Complexity

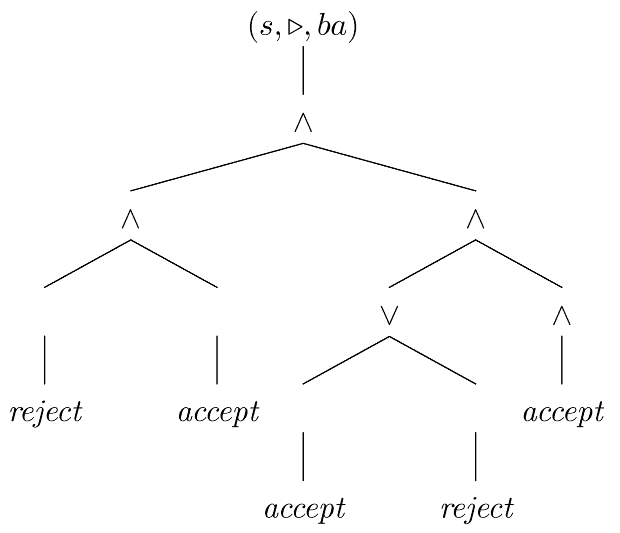
Like for deterministic and nondeterministic Turing machines we define alternating time and space.

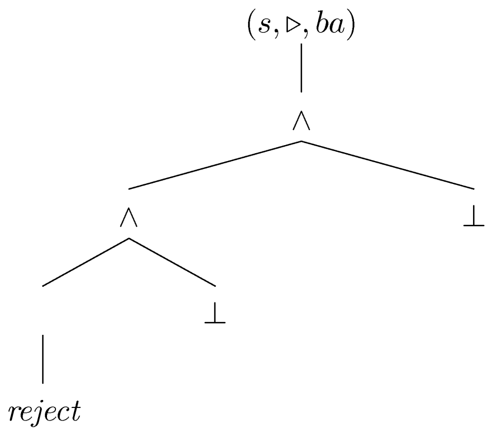
To get a good measure of resources an alternating machine needs for a computation we should not count configurations not necessary for deciding the input. To decide which configurations have an impact on the complexity we use the labeling function $l$.

**Definiton 2.4.9** (ATIME)**.** *Let $A$ be an alternating Turing machine. $A$ accepts an input $x$ in time $t$ if there is a labeling* l *satisfying the Conditions 2.1 with* $\mathrm{l}((s, \triangleright, x, \triangleright, \epsilon, \dots, \triangleright, \epsilon)) = 1$ *and for every configuration $C$ it holds that* $\mathrm{l}(C) \neq \perp$ *implies* $\exists k < t : (s, \triangleright, x, \triangleright, \epsilon, \dots, \triangleright, \epsilon) \xrightarrow{A^k} C$.


$(s, \triangleright, ba)$
$(r, \triangleright b, a)$
$(r_b, \triangleright b, a)$
$(r, \triangleright ba, \epsilon)$
$(c_a, \triangleright, ba)$
$(c_a, \triangleright ba, \epsilon)$
$(r_a, \triangleright ba, \epsilon)$
$(r, \triangleright ba\sqcup, \epsilon)$
$(\text{“no”}, \triangleright, ba)$
$(\text{“yes”}, \triangleright ba, \epsilon)$
$(c_b, \triangleright b, a)$
$(c_b, \triangleright ba\sqcup, \epsilon)$
$(\text{“yes”}, \triangleright ba\sqcup, \epsilon)$
$(\text{“yes”}, \triangleright b, a)$
$(\text{“no”}, \triangleright ba\sqcup, \epsilon)$

(a) The computation tree





(b) tree for naive evaluation (c) tree for short-circuit evaluation

**Figure 2.4:** Computation tree of the computation with input $ba$ on the alternating Turing machine from Example 2.4.7.

So an alternating Turing machine may accept an input in linear time even if there are some infinite computation paths.

As for deterministic and nondeterministic Turing machines we want to define space complexity for alternating Turing machines. Like for time complexity we exclude the configurations not necessary for deciding the input.

**Definiton 2.4.10** (ASPACE)**.** *Let $A$ be an alternating Turing machine. $A$ accepts an input $x$ in space $s$ if there is labeling* $\mathrm{l}$ *sufficing the conditions 2.1 with* $\mathrm{l}((s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon)) = 1$ *and for every configuration $C$ in the range of* $\mathrm{l}$ *holds that* $\mathsf{SPACE}(C) \leq s$*.*

**Complexity Classes**

We know define some alternating complexity classes that are closed under reductions.

**Definiton 2.4.11** (ALOGSPACE). *The class* ALOGSPACE *(alternating logarithmic space) is the class of problems decided by an alternating Turing machine in logarithmic space.*

$$\mathsf{ALOGSPACE} = \mathsf{ASPACE}(\log n)$$

**Definiton 2.4.12** (AP). *The class* AP *(alternating polynomial time) is the class of problems decided by an alternating Turing machine in polynomial time.*

$$\mathsf{AP} = \bigcup_{j>0} \mathsf{ATIME}(n^j)$$

**Definiton 2.4.13** (APSPACE). *The class* APSPACE *(alternating polynomial space) is the class of problems decided by an alternating Turing machine in polynomial space.*

$$\mathsf{APSPACE} = \bigcup_{j>0} \mathsf{ASPACE}(n^j)$$

**Definiton 2.4.14** (AEXPTIME). *The class* AEXPTIME *(alternating exponential time) is the class of problems decided by an alternating Turing machine in exponential time.*

$$\mathsf{AEXPTIME} = \bigcup_{j>0} \mathsf{ATIME}(2^{n^j})$$

## 2.5 Simulating alternating Turing Machines

In this section we focus on the relations between alternating complexity classes and deterministic complexity classes. Our overall goal is writing alternating programs that can be executed efficiently on a common computer. So we are interested in alternating complexity classes that are contained in feasible deterministic classes.

**Theorem 2.5.1.** *Chandra et al., 1981 Let $T(n) \geq n$ be a function computable in time $O(T(n))$.If an alternating Turing machine $M$ accepts in time $T(n)$ then there exists an alternating Turing machine $N$ with $L(M) = L(N)$ and all computation paths of $N$ are at most of length $O(T(n))$*

*Proof.* Let $M$ be a $k$-tape alternating Turing machine satisfying the conditions of the theorem. Construct the following $k+1$-tape alternating Turing machine $N$:
$N$ first computes $T(n)$ on the additional tape. Then $N$ simulates $M$ and decreases the counter on the additional tape in every computation step of $M$. If the counter equals 0 and $M$ is not in a halting configuration then $N$ rejects.

Clearly all computation paths of $N$ are at most of length $O(T(n))$. It remains to show that $N(x) = M(x)$ for all possible inputs $x$. Let $C_N, C_M$ be the sets of configurations from $N(x)$ and

$M(x)$. After the computation of $T(n)$ each configuration in $N$ can be written as tuple $(c_i^M, t)$ with $c_i^M \in C_M$ and $t$ is an integer with $0 \leq t \leq t(n)$. By the definition of $N$ it holds that:

$$(c_i^M, t) \xrightarrow{N} (c_j^M, t-1) \iff c_i^M \xrightarrow{M} c_j^M$$

The computation of $T(n)$ is deterministic and therefore the machine $N$ accepts an input if and only if the labeling $l$ deciding the input suffices $l((c^M, t)) = 1$ for $c^M$ the initial configuration of $M$.

As $M$ works in time $T(n)$ there is a evaluation tree that only uses configurations reachable in at most $T(n)$ step from the initial configuration. So all configurations of $M(x)$ are also represented in the evaluation tree of $N$. The difference in the two evaluations is the labeling of configurations with distance $T(n)$ to the initial configuration $c^M/(c^M, T(n))$. $M$ labels such configurations with $\perp$ while $N$ labels them with 0.

To prove that these evaluations are equal, we define the an equivalence relation $\{\{1\}, \{\perp, 0\}\}$, that is that $0 \equiv \perp$. It is just a calculation that the $\wedge$ and $\vee$ operators preserve this equivalence relation and therefore $N(x) \equiv M(x)$. We notice the fact that a labeling for $N$ only uses the values $\{0, 1\}$ and a deciding labeling for $M$ labels $c^M$ with either 0 or 1. Combing this with $N(x) \equiv M(x)$ we get $N(x) = M(x)$. □

**Theorem 2.5.2.** *Chandra et al., 1981 Let $S(n) \geq \log n$ be a function computable in space $O(S(n))$. Let be $M$ an alternating Turing machine accepting in space $S(n)$. Then there exist an alternating Turing machine $N$ with $M(x) = N(x)$ and the following hold for $N$. All computation paths of any input are at most of length $O(c^{S(n)})$, for some constant $c$. Every configuration $C$ reachable from the initial configuration satisfies* $\mathsf{SPACE}(C) \leq S(n)$.

*Proof.* Let $M$ be a $k$-tape alternating Turing machine satisfying the conditions of the theorem. We define a constant $c' = |S \cup \{\text{“yes”}, \text{“no”}\}| \cdot |\Sigma|$. The number of configurations necessary for deciding the acceptance is bounded by $c'^{S(n)} \cdot S(n)^{k-1} \cdot n =: c^{S(n)}$ and therefore $M$ accepts in time $c^{S(n)}$.

Construct the $k+1$-tape alternating Turing machine $N$:
$N$ initially writes $c^{S(n)}$ on the additional tape. To keep in the space bounds the machine uses a c-ary representation. On the other tapes $N$ writes a tape end symbol after $S(n)$ symbols. $N$ simulates $M$ and decrements the counter an the additional tape for every computation step of $M$. If the counter goes down to 0 or one of the tapes reaches the tape end symbol $N$ rejects.

It is clear that $N$ suffices the theorem's time and space bounds. What is remained is to show that $N(x) = M(x)$ for all possible inputs $x$. This is done in the same way as in the proof of Theorem 2.5.1. □

Whenever we deal with alternating Turing machines we can assume that these machines have no infinite paths, simple using Theorem 2.5.1 and Theorem 2.5.2. This is very useful for theoretic purposes but we show later that the counter technique used in the proofs can be very awful in practice.

Chandra et al., 1981 developed very strong relations between alternating and deterministic complexity classes, which are useful to identify alternating programs that can be simulated efficiently:

**Theorem 2.5.3.** *Chandra et al., 1981*

$$\mathsf{ATIME}(f(n)) \subseteq \mathsf{SPACE}(f(n)) \qquad f(n) \geq n$$

**Theorem 2.5.4.** *Chandra et al., 1981*

$$\mathsf{ASPACE}(f(n)) = \bigcup_{c>0} \mathsf{TIME}(c^{f(n)}) \qquad f(n) \geq \log n$$

*Proof sketch.* The most interesting part for our purposes is the technique used in the reduction from alternating space to deterministic time, because we want to reuse it for our framework. So we bring the main proof ideas for the following equation and refer the reader to Chandra et al., 1981 for the other direction.

$$\mathsf{ASPACE}(f(n)) \subseteq \bigcup_{c>0} \mathsf{TIME}(c^{f(n)}) \qquad f(n) \geq \log n \tag{2.2}$$

Let $A$ be an alternating Turing machine working in space $f(n)$. The deterministic Turing machine $M$ works as follows:

- $M$ computes $f(n)$
- $M$ writes down all possible configurations $C$ with $\mathsf{SPACE}(C) \leq f(n)$ (these are at most $b^{f(n)}$ for some constant b)
- label each of this configurations with a $\perp$
- compute a labeling which fulfills Conditions 2.1 by iterating the rules in 2.1 for computing the new labels. (we need at most $b^{f(n)}$ iterations)
- return the label of the initial configuration

So $M$ works on an tape of length $O(S(n) \cdot b^{f(n)})$ and read it at most $b^{f(n)}$ times. We can simplify these bounds to: $M$ works in time $c^{f(n)}$ for a new constant $c$. □

So the main idea in the proof is to save configurations and their leading to acceptance value. This prevents the algorithm from computing a configuration leading to acceptance value several times, when the configuration occurs multiple times in the computation tree.

**Corollary 2.5.5.** *Chandra et al., 1981 Using the Theorems 2.5.3, 2.5.4 and the fact that* QSAT *is in* AP *we get the following identities:*

$$\begin{aligned} \mathsf{EXPSPACE} &= \mathsf{AEXPTIME} \\ \mathsf{EXPTIME} &= \mathsf{APSPACE} \\ \mathsf{PSPACE} &= \mathsf{AP} \\ \mathsf{P} &= \mathsf{ALOGSPACE} \end{aligned}$$

Recall that P is a class of feasible computable problems and a problem hard for one of the classes PSPACE, EXPTIME, EXPSPACE is considered as inherent hard problem. So we identify ALOGSPACE as the class of feasible computable alternating problems.

## 2.6 Natural Alternating Problems (in the class P)

In the last section we mentioned that the complexity classes ALOGSPACE and P are equivalent and therefore identified the class ALOGSPACE as the class of feasible computable alternating problems. Further there are many problems known to be in the class P and therefore many problems that can be computed with an alternating program in a feasible way.
But in general there is no benefit in solving P problems with an alternating machine and then simulate this alternating machine. In fact for many problems it's much harder to find an alternating algorithm than a deterministic one and even if we find some, it would be a counterintuitive algorithm with much more code to write.

So alternation is not the right solution for all problems in P but there are some problems that can be solved by alternating machines in a very natural and much easier way than with traditional algorithms. In this section we present some problems in the class P which can be solved very intuitively with an alternating algorithm.
We identified such problems using the collections of P-complete problems in Greenlaw et al., 1995 and Miyano et al., 1989, so all problems in this section can be found there.

### 2.6.1 HORN problems

The first group of problems comes from propositional logic. We are interested in properties of so called Horn formulas, so we start with the definition of Horn clauses and Horn formulas.

**Definiton 2.6.1** (Horn clause)**.** *We call a formula Horn clause iff $\varphi$ is a clause (disjunction of literals) with at most one positive literal.*

We distinguish three different types of Horn clauses. The clauses with a positive literal $p$ and some negative literals $\neg q_1, \dots . \neg q_n$. They are equivalent to a formula $q_1 \wedge \cdots \wedge q_n \rightarrow p$ and so we call them rules. Clauses which only consist of a positive literal are facts. Clauses with negative literals but no positive literals are called constraints.

**Definiton 2.6.2** (definite Horn clause)**.** *We call a formula definite Horn clause iff $\varphi$ is a clause with exactly one positive literal.*

A definite Horn clause is either a fact or a rule.

**Definiton 2.6.3** (Horn formula)**.** *We call a formula $\varphi$ (definite) Horn formula iff $\varphi$ is in conjunctive normal form and each clause is in (definite) Horn form.*

### Definite Horn Minimal Model

***Given:*** A define Horn formula $\varphi$ and a atom $p$.

***Problem:*** Is $p$ in the minimal model of $\varphi$

So $p$ is in the minimal model if p is a fact in $\varphi$ or $p$ is the positive literal in a clause and all other atoms occurring in the same clause are in the minimal model. The case that $p$ is a fact is just a special case of the second one where $p$ is the only literal in the clause.

**Alternating Approach 2.6.4** (Definite Horn Minimal Model)**.** *$p$ is in the minimal model if there* EXISTS *a clause $C \in \varphi$ containing $p$ as literal such that* FORALL *negative literals $\neg q_i \in C$ it holds that $q_i$ is in the minimal model. Checking that $q_i$ is in the minimal model is done in the same way.*

This algorithm terminates when arriving at a clause $C$ that is a fact. For fact $C$ there are no negative literals and therefore no successors. It is common to interpret universal quantifiers over the empty set as accept and existential quantifiers over the empty set as reject.

A problem in this algorithm is that we may get some cycles. For example if $\varphi$ includes the rules $q \rightarrow p, p \rightarrow q$ asking for $p$ would lead to infinite computation path alternating this two rules. This can be resolved by introducing a counter that is incremented for every clause we use and rejecting when we exceed the total number of clauses. In a computation path that exceeds this counter at least one literal is computed twice and therefore we have a cycle in the path. We present a better solution to get rid of such cycles later.

As we already mentioned alternating complexity classes are closed under complement and therefore we can give an alternating algorithm for the co-problem.

**Alternating Approach 2.6.5** (co-Definite Horn Minimal Model)**.** *$p$ is not in the minimal model if* FORALL *clauses $C \in \varphi$ containing $p$ as literal there* EXISTS *a negative literal $\neg q_i \in C$ such that $q_i$ is not in the minimal model. Checking that $q_i$ is not in the minimal model is done in the same way.*

We now use this algorithm to solve the satisfiability problem for Horn clauses.

### Horn Satisfiability (HORNSAT)

***Given:*** A Horn formula.

***Problem:*** The task is to decide if the clauses are satisfiable or not.

*Ref:* Kasif, 1990, Plaisted, 1984

The satisfiability problem of general propositional logic formulas is a famous NP-complete problem.

A Horn formula is a definite Horn formula plus some constraint clauses. A deterministic algorithm would compute the minimal model for the definite Horn part and then check all constraints with this model. We reuse this concept for our alternating approach but the other way round. In the first step we branch over the constraints and then we check them.

**Alternating Approach 2.6.6** (Horn Satisfiability)**.** *$\varphi$ is satisfiable if* FORALL *constraint clauses $C \in \varphi$ there* EXISTS *a negative literal $\neg q_i \in C$ such that $q_i$ is not in the minimal model. For Checking that $q_i$ is not in the minimal model we use the Alternating Approach 2.6.5.*

### 2.6.2 Games

A very common class of natural alternating problems are two player games, more precisely the problem of finding a winning strategy in such a game. For the most games this problem is much harder than P, its well known that Go (with Japanese rules) is EXPTIME completeFraenkel and Lichtenstein, 1981, Robson, 1983 and even the simplified version in Papadimitriou, 1994, which forbids exponentially long games is PSPACE-complete. But if every configuration of the game can be encoded efficiently (means in SPACE$(\log n)$), we can decide the existence of a winning strategy in P.
We start with a simple two player game.

#### Two player Game (GAME)

***Given:*** A two player game G.
This is a tuple $G = (P_1, P_2, W_1, s, M)$ with $P_1 \cap P_2 = \emptyset$, $W_1 \subseteq P_1 \cup P_2$, $s \in P_1$ and $M \subseteq P_1 \times P_2 \cup P_2 \times P_1$.

- $P_1$ is the set of game position where it is Player 1's turn and $P_2$ is defined similar.
- $W_1$ the set of immediate winning positions for player 1
- $s$ the starting position
- $M$ is the set of allowed moves. So if a player is in position $p$ and $(p, q) \in M$ the player may move to position $q$. The definition of $M$ ensures that the turns alternate between the two Players.

We define a winning position with a kind of recursion. A position $p$ is called a winning position if and only if one of the following conditions holds:

- $p \in W_1$ (p is a immediate winning position)
- $p \in P_1$ and there exists a move $(p, w) \in M$ such that $w$ is a winning position
- $p \in P_2$ and for every move $(p, w) \in M$ holds that $w$ is a winning position

***Problem:*** Is s a winning position for Player 1 ?
*Ref:* Jones and Laaser, 1974, Greenlaw et al., 1995

It is straightforward to solve this with an alternating approach.

**Alternating Approach 2.6.7** (Two player Game)**.** *Player 1 has a winning strategy in the Game G if there* EXISTS *a move for player 1 such that* FORALL *possible moves of player 2 player 1 has a winning strategy*

The definition of GAME captures almost all two player games, but for the most games the encoding as GAME would lead to an exponential or higher blow up of the input. It is quite intuitive that GAME captures deterministic games where 2 players alternate moving but it even captures many games with nondeterministic elements like rolling dices. In our definition of winning strategy we make no assumption about the strategy of player 2. So player 2 does not try to optimize something he just randomly picks out one possible move and our strategy for player 1 has to handle all possible moves. To encode nondeterministic games we just encode the dicing in the possible moves for player 2.

But for nondeterministic games the question for a winning strategy is inferior because the answer would be no in the most cases. For nondeterministic games we would need a strategy that optimizes the winning probability, but we do not mention such problems here.
With a similar technique we can reduce the problem of finding a winning strategy in n-player games to Two player Game. We just encode each possible combination of moves from player $2, 3, \dots, n$ as move for player 2 in the Two player Game.

In the following we present some games, with simple configurations that can be saved in $\mathsf{SPACE}(\log n)$ such that the problem of finding a winning strategy is in the class $P$.

### Cat and Mouse(CM)

***Given:*** A directed graph $(V, E)$. Vertices $c, m, g \in V$
The game is played as follows: The cat starts on vertex c, the mouse on m and vertex g is the goal (the mouse hole). The mouse has the first move and then moves alternate between cat and mouse. In each move the player (cat or mouse) has the choice to follow an edge in the graph or stay at the current vertex. The mouse loses if it gets caught by the cat, which means that mouse and cat occupy the same vertex. The mouse wins if it reaches the mouse hole without being caught.
***Problem:*** Does the mouse have a winning strategy?
*Ref:* Chandra et al., 1981

**Alternating Approach 2.6.8** (Cat and mouse)**.** *The mouse has a winning strategy if there* EXISTS *a move for the mouse such that* FORALL *possible moves of the cat, the mouse has a winning strategy*

### Acyclic Geography Game (AGG)

***Given:*** An acyclic directed graph $G = (V, E)$ and a vertex $s$. The game is played as follows:

We start with a token on the vertex $s$. Player 1 has the first move and then the players alternate moving. In each move the active player can move the token along one edge. The first player with no possible move loses.
***Problem:*** Does Player 1 have a winning strategy?
*Ref:* Chandra and Tompa, 1990

**Alternating Approach 2.6.9** (Acyclic Geography Game)**.** *Player 1 has a winning strategy starting in $s$ if there* EXISTS *an edge $(s, v)$ such that* FORALL *edges $(v, v')$ there is winning strategy for player 1 starting in $v'$.*

### Two-person Pebble Game Fixed Rank(PEBBLE)

***Given:*** A fixed rank pebble game. A pebble Game of rank k is a tuple $G = (X, R, S, t)$ with:

- $X$ a finite set (of pebbles)
- $R \subseteq \{(x, y, z)|x, y, z \in X, x \neq y, x \neq z, y \neq z\}$ the set of rules
- $S \subseteq X$ the start set of size k
- $t \in X$ the terminal pebble

The Pebble Game is played as follows:
At the beginning the pebbles of $S$ are on the board $B := S$. Then the players alternate (starting with player 1) replacing one pebble from the board by another pebble from the pool $X$ according to the rules. The rule $(x, y, z)$ says that if the pebbles $x, y$ are on the board $(x, y \in B)$ and $z$ is not $(z \notin B)$ the active player can replace $x$ by $z$.
The player who pebbles $t$ or gets the opponent in a no move position wins.
***Problem:*** Does Player 1 have a winning strategy?
*Ref:* Kasai et al., 1979

**Alternating Approach 2.6.10** (Two-person Pebble Game Fixed Rank)**.** *Player 1 has a winning strategy in the pebble Game $G$ starting with $S$ if there* EXISTS *a rule $(x, y, z)$ with $x, y \in S, z \notin B$ such that* FORALL *rules $(x', y', z')$ with $x', y' \in \{S\backslash\{x\}\}\cup\{z\}, z' \notin \{S\backslash\{x\}\}\cup\{z\}\}$ there is winning strategy for player 1 with start set $S' = \{\{S \setminus \{x\} \cup \{z\}\} \setminus \{x'\}\} \cup \{z'\}$.*

## 2.6.3 Circuits

We start with a family of circuits restricted to AND and OR gates:

### Monotone Circuit Value Problem (MCVP)

***Given:*** A monotone Boolean circuit $\alpha$ (a circuit built only of AND and OR gates) with inputs $x_1, \dots, x_n$ and output $y$
***Problem:*** Is output $y$ true on the inputs $x_1, \dots, x_n$
*Ref:* Greenlaw et al., 1995

**Alternating Approach 2.6.11** (Monotone Circuit Value Problem)**.** *An AND gate is true if* FORALL *predecessor gates it holds that they are true and false otherwise. A OR gate is true if there* EXISTS *a predecessor gate which is true and false when no such predecessor exists. An input gate has the true value of his corresponding input. To compute the value of $y$ we just use these rules.*

**Alternating Approach 2.6.12** (co-Monotone Circuit Value Problem)**.** *An AND gate is true if there* EXISTS *a predecessor gate that is true and false otherwise. A OR gate is true if* FORALL *predecessor gates it holds that they are true and false otherwise. An input gate has the negated true value of his corresponding input. To compute the value of $\neq y$ we just use these rules.*

Now we use these rules for solving the Circuit Value Problem for general boolean circuits, we add negating gates to the circuits.

### Circuit Value Problem (CVP)

***Given:*** A Boolean circuit $\alpha$ with inputs $x_1, \dots, x_n$ and output $y$
***Problem:*** Is output $y$ true on the inputs $x_1, \dots, x_n$
*Ref:* Ladner, 1975

**Alternating Approach 2.6.13** (Circuit Value Problem)**.** *To compute the true value of $y$ We start using the rules from the Alternating Approach2.6.11 for AND and OR gates. When evaluating a NOT gate then we skip to the rules for the successors from the rules of the Alternating Approach 2.6.11 to the rules of Alternating Approach 2.6.12 or the other way round.*

### Alternating Monotone Fanin 2, Fanout 2 Circuit Value Problem (AM2CVP)

***Given:*** A monotone Boolean circuit $\alpha$ with inputs $x_1, \dots, x_n$ and a output $y$ with the following properties: The gates alternate between AND and OR on every path from a input to the output. Inputs are only connected to OR gates. The gates have fanout exactly 2 (except output gates) and fanin 2 except the input gates. $y$ is an OR gate.
***Problem:*** Is output $y$ true on the inputs $x_1, \dots, x_n$
*Ref:* Greenlaw et al., 1995

The class of Alternating Monotone circuits leads to a very simple alternating algorithm:

**Alternating Approach 2.6.14** (Alternating Monotone Circuit Value Problem)**.** *The output $y$ is true if there* EXISTS *a predecessor gate $p$ such that* FORALL *predecessors $p_i$ of $p$ it holds that they are true. All these $p_i$ are OR gates or input gates. In the first case we use the same alternating procedure as for $y$ in the other case we just took the true value from the corresponding input.*

### 2.6.4 Graphs

We saw some games played on graphs as alternating problems, but there are still more graph problems that can be solved with alternating algorithms. As example we bring the alternating graph accessibility problem.

**Alternating Graph Accessibility Problem (AGAP)**

***Given:*** A directed Graph $G = (V, E)$, with $V = A \cup B,\ A \cap B = \emptyset$ and two vertices $s, t$,
The relation $\text{apath}(x, y)$ on $G$ holds if one of the following conditions is satisfied:

- $x = y$
- $x \in B$ and $\exists z \in V, (x, z) \in E$ such that $\text{apath}(z, y)$
- $x \in A$ and $\forall z \in V, (x, z) \in E$ such that $\text{apath}(z, y)$

***Problem:*** Is $\text{apath}(s, t)$ true
*Ref:* Chandra et al., 1981,Immerman, 1981,Immerman, 1987

**Alternating Approach 2.6.15** (Alternating Graph Accessibility Problem). *if* $s = t$ $\text{apath}(s, t)$ *holds otherwise we check* $\text{apath}(s, t)$ *with an alternating approach.*
*If* $s \in A$ $\text{apath}(s, t)$ *is true if* FORALL *neighbors* $z, (s, z) \in E$ *holds* $\text{apath}(z, y)$. *Otherwise if* $s \in B$ $\text{apath}(s, t)$ *is true if there* EXISTS *a neighbor* $z, (s, z) \in E$ *with* $\text{apath}(z, y)$. *The true value from* $\text{apath}(z, y)$ *is computed in the same way.*

## 2.7 The Computation Tree

Using alternating Turing machines we get yes and no answers, for example there is a winning strategy or not. Often we are interested in more details. We want to know which strategy is a winning strategy and not just that there exists one (Mathematicians happy with existence statements may skip this section). When using deterministic Turing machines there is the tape content of the halting configuration that gives additional information about the solution. An alternating Turing machine in generall has many leaves in its computation tree, so we can't pick the information just from one configuration. We have to consider all configurations in the computation tree and also the edges between them to get additional information about the solutions.

We remind the Definitions 2.4.4, 2.4.6 and bring some examples of how information can be encoded in the computation tree of an alternating machine with some input.

### 2.7.1 Proof Trees in Horn Problems

When computing that $p$ is in the minimal model of a definite Horn formula $\varphi$ we may also be interested in proving this. We can use the computation tree as proof tree. To prove that $p$ is in the minimal model we have to find a rule $q_1, \ldots, q_n \to p,\ n \geq 0$ and proofs for $q_1, \ldots, q_n$.

There are two types of configurations in the computation tree. First the existentially quantified configurations with literals to be proven and then the universal configurations with rules to be satisfied.

To get a proof that $p$ is in the minimal model we start with the configuration representing $p$ (the initial configuration) and then follow the (first) leading to acceptance successor configuration. This successor configuration represents a rule of the form $q_1, \dots, q_n \rightarrow p,\ n \geq 0$ and has the successor configurations representing $q_1, \dots, q_n$ and all of them leads to acceptance.

Assuming we have a proof for $q_1, \dots, q_n$ are in the minimal model we can construct the proof for $p$ is in the minimal model by simple adding $q_1, \dots, q_n \rightarrow p$ to the proof. The proofs for the $q_i$ can be computed recursively from the computation tree with the same idea. This can be formally proven by induction on the tree size and the fact that every accepting computation path is a finite path.

We mention here that in general it is not possible to compute the minimal model from the computation tree. There may be facts and rules in $\varphi$ that the alternating machine $A$ has not used to compute that $p$ is in the minimal model.

### 2.7.2 Winning Strategy in Games

The computation tree of a game problem saves game configurations, i.e. the position of tokens on the game board, in the machine configurations. The difference between two connected configurations is the move made by one player. Machine configurations of a winning strategy machine can be distinguished in two sets. The existentially quantified moves where player 1 has to make a move and the universally quantified moves where its player 2 turn.

We can identify the first move of a winning strategy by choosing a leading to acceptance successor $A$ of the initial configuration. Now the opponent makes a move which leads to a successor $E$ of $A$. $A$ was universal quantified and therefore $E$ is a leading to acceptance configuration. Therefore there is a leading to acceptance successor for $E$ which gives us the next move for the winning strategy. We do this until there is an immediate winning position. For a formal proof once more we can use induction and the fact that an accepting computation path is finite.

### 2.7.3 Circuit Value

An alternating Turing machine evaluating circuit value problems would save the current gate of the evaluation in the configuration. So here we can use the computation tree to get the true values of some internal gates or find all gates that were evaluated to compute the true value of the output gate. A gate is true if it is encoded in a leading to acceptance configuration (it is labeled with 1) and false if the configuration is labeled with 0. We can't make a decision about gates not encoded in a configuration or encoded in a configuration labeled with $\perp$.

#### Chapter 3

# A Java extension for the alternation paradigm

In the last chapter we introduced the concept of alternation in form of alternating Turing machines. Further we gave a method how to simulate alternating Turing machines on deterministic ones. The question left is how we can use this concepts for programming in practice.

To get an answer for this question we introduce a language, based on Java syntax, to write alternating programs. Sure it does not suffice to get a syntax for programming, we further want to compute the output of such a program. So we present a framework that simulates the alternating algorithms written in our language.

## 3.1 Language Specification

Following ideas from alternating Turing machines we introduce a programming language for alternating algorithms. For specifying the syntax of alternating programs we use ANTLR grammar syntax Parr, 2007. To keep the big picture, we sometimes skip some details of the syntax but a full grammar for our alternating language is attached as Appendix A.

### 3.1.1 Idea of Alternating Programming

The main idea of alternating programming, following the concept of alternating Turing machines, is to define states which compute successor states and combine the successors results to a return value for the primal state. Following the idea of alternating Turing machines we have further to distinguish between input variables and work variables. Input variables are variables that the alternating algorithm can not modify. We give a more precise specification later. Work variables are variables that describe the configurations of an alternating program.

According to the previous comments, an alternating program is structured as follows. At the beginning there is the place for header information. The program starts with the *atm* keyword and the programs name. After the name there is the place to define the input variables for the alternating algorithm. Then there is a block with the definition of our work variables and the code for the states, the central part of an alternating program.

```
header
'atm' Identifier  '('input')' {
        worktape
        state+
}
```

**Listing 3.1:** syntax: alternating program

An alternating program works like this: The machine initializes the work variables and starts the computation in the first state. A state makes some computation using input and work variables and then either immediately accepts / rejects the input or computes a quantifier and a set of successor configurations. A configuration in an alternating program is defined analogously to alternating Turing machines, so a configuration is simply a state with an instance of the work variables. A configuration that does not immediately decide the input, has a successors set and a quantifier. Such a configuration accepts the input if one or all successors, according to the quantifier, accepts.

### 3.1.2 Input Variables

As we already mentioned it is necessary to distinguish between input variables and work variables in our alternating programs. So input variables basically are variables we should not modify, but for using arbitrary objects we have to be a bit careful. Sure it should be possible to call methods of input objects, but on the other hand a method call may change the object properties. There are also some changes in the object that would not violate the input characterization. So it would be ok to initialize an attribute of the object when we access it the first time. So we have to define precisely what we expect the programmer to do with input objects and what not to do.

**Postulation 3.1.1** (input variables)**.** *In our framework the access on input variables is restricted to the following actions:*

- *read access to primitive data type variables*
- *read access to objects variables*
- *"read method" calls*

*A read method is a method that returns the same value whenever we call it with the same parameters. Furthermore such a method must not influence the results of other read methods or Java primitive variables we use in our alternating program.*

So calling an ordinary get-method that just hands over a variable is ok. It's also allowed to use a method computing its return value deterministically out of the objects attributes. The second condition causes that the read method property is not an independent property of one method but of the collection of methods used in the program. For example we assume a get-method that increments a counter at every access to the variable. We can use this get-method for input variables in alternating programs as long as we do not access the counter or methods using the counter.

We define the programs input variables simply as a list of type-name pairs. The syntax is described in Listing 3.2.

```
type Identifier (',' type Identifier )*
```

**Listing 3.2:** syntax: input

Finally we give an example of an input definition:

```
atm GGAtm (Graph.Node<GraphGame.Position> startPlayer,
                        Graph<Position> gameBoard)
```

**Listing 3.3:** example for a head of an alternating program

### 3.1.3 Work Variables

With work variables we mean the variables representing the configurations of our machine. This is almost equivalent to the variables used for computations in the states but there are some cases where local variables can be used for computations. We come back to these local variables later. As the work variables represent the configurations, they also determine the number of possible configurations. To prevent too bad running time the work variables have to be restricted to something like $\mathsf{SPACE}(\log n)$.

According to $\mathsf{SPACE}(\log n)$ Turing machines we can use counters and cursors in our Turing machine. So we can use primitive data types for counting or saving some flags and variables with an arbitrary Object-Type for cursors on the input.

**Postulation 3.1.2** (work variables)**.** *There are two different kind of work variables. Those storing data that can be modified and those holding a reference to an input object.*

***Data variables:*** *These are variables having one of the following primitive data types 'boolean', 'char', 'byte', 'short', 'int', 'long', 'float', 'double'. In alternating algorithms any value of the specified type, can be assigned to such variables. For technical reasons all such variables must be initialized before the alternating computation starts.*

***Cursor variables:*** *Variables with object type are handled as cursor variables. This means an alternating algorithm can't assign arbitrary "values" to such variables. So an alternating program can't create new objects and assign them to variables. The only valid way a cursor variable can be used is storing an object from the input or the 'null' value.*

***Arrays:*** *A special case is the one-dimensional array. As arrays are objects they should be cursor variables, but to get the full power of alternating Turing machines we use them as data variables. When using an array as work variable we first have to restrict us to one-dimensional* *arrays. The array must be initialized before the alternating computation starts. The elements of an array are handled according to their type. So an array of object types holds references to the input and an array of primitive types holds values.*

*We can't use an array as cursor to an input array, if we need such a functionality we have to use other data types like* java.util.List.

We mention a graph problem, where the graph $G$ is part of the input. For counting the edges already used for the computation we would use a data variable of type int or long. For saving the current position in the graph we would use a cursor variable of an appropriate object type. When solving graph games where the number of tokens may vary between the instances an array would be an appropriate way to store the tokens positions.

The listings 3.4 and 3.5 give the syntax for defining the work variables. This syntax guarantees that primitive variables and arrays are initialized.

```
declaration*
```

**Listing 3.4:** syntax: worktape

```
classOrInterfaceType Identifier (',' =Identifier)* ';'
| primitiveType Identifier '=' expression ';'
| classOrInterfaceType Identifier '=' expression ';'
| oneDimensionalArrayType Identifier '=' 'new' arrayCreator ';'
```

**Listing 3.5:** syntax: declaration

We complete this section with an example of a work tape definition.

```
Graph.Node<Integer> catPosition=catStart;
Graph.Node<Integer> mousePosition=mouseStart;
int counter=0;
```

**Listing 3.6:** example for a work variable declaration in an alternating program

### local variables

Often we need variables to iterate over some collections or arrays. As these variables have no descriptive statement about the configuration we should not define them as work variables. So there is the option to define local variables in For and For-Each loops.

```
for (int i=0 ; i<=2*r; i++){
        [...]
}
```

**Listing 3.7:** For loop with local variable

```
for(Graph.Node child: mousePosition.getChildren()){
        [...]
}
```

**Listing 3.8:** For–Each loop with local variable

### 3.1.4 States

States are the central part of an alternating program, the part where the alternating algorithm is written. A state definition begins with the state keyword followed by the states name and a code block.

```
'state' Identifier {
        statement+
}
```

**Listing 3.9:** syntax: state

In alternating programs a state usually starts with an ordinary deterministic computation. For this computation the programmer can use most of the Java constructs. An important restriction is that the programmer is not allowed to define local variables, except the mentioned special cases. Further the programmer has to care about the input objects; they should not be modified by the program.

Further there are some restrictions on object-oriented features. So there is no *new* command, in alternating programs, since no new objects should be created. As an alternating program is no ordinary class there are no methods, class or interface definitions and no keywords *this* or *super*. Listing 3.10 gives the definition of the allowed Java statements.

```
  block
| 'if' parExpression statement (options {k=1;}:'else' statement)?
| 'for' '(' forControl ')' statement
| 'while' parExpression statement
| 'do' statement 'while' parExpression ';'
| 'try' block catches 'finally' block
| 'switch' parExpression '{' switchBlockStatementGroups '}'
| 'throw' expression ';'
| 'break' (Identifier)? ';'
| 'continue' (Identifier)?  ';'
| ';'
| statementExpression ';'
| Identifier ':' statement
```

**Listing 3.10:** syntax: statement part1

To enable alternation there are some additional constructs in an alternating program. First there are the commands *accept*, *reject* which immediately decide a configuration. The *accept* command says that this configuration is a leading to acceptance configuration and the *reject* command tells that the configuration does not lead to acceptance.
The alternation in the algorithms comes from the *forall* and *exists* statement. The *forall* command says that the configuration has a set of successors (maybe the empty set) and it only leads to acceptance if all successors lead to acceptance. The *exists* statement has similar semantics, the configuration leads to acceptance if one successor leads to acceptance. As the semantics of the *forall* and *exists* commands needs the successor configurations they are always followed by a block defining these successors.

```
| 'accept' ';'
| 'reject' ';'
| 'forall' '{' nondeterminism '}'
| 'exists' '{' nondeterminism '}'
```

**Listing 3.11:** syntax: statement part2

When programming functions every valid computation has to end with a return statement. Similarly this holds for states in alternating programs. Every possible computation must end either with an immediate decision of the input (accept/reject) or with an nondeterminism (forall/exists).

```
state alternatingTuringMachine {
        if(stateType.get(activeState).equals('A')) {
                accept;
        }
        if(stateType.get(activeState).equals('F')) {
                forall{
                        [...]
                }
        }
        if(stateType.get(activeState).equals('E')) {
                exists{
                        [...]
                }
        }
        reject;
}
```

**Listing 3.12:** An example for a state in an alternating program (simplified)

### Forall / Exists Blocks

The thing still missing on our alternating programs is the computation of the successors. So a nondeterminism block after the *forall* or *exists* statement has to define new configurations. For a configuration we need a state and work variables. So a nondeterminism block is basically a list of state, work variables pairs. Such a pair is the name of the state and a block defining the work variables. Sometimes the number of successors depends on the input. In such cases alternating programs uses for loops to compute the successors.

```
(Identifier '{' stateStatement* '}' | stateLoop)+
```

**Listing 3.13:** syntax: nondeterminism

We can use all the Java constructs defined in 3.10 to modify the work variables for the new configuration. It is important to mention that the computations for the different configurations are independent of each other. So each computation starts with the work variables as they are set when the computation enters the nondeterminism block.

```
'for' '(' forControl ')' '{' stateCall '}'
```

**Listing 3.14:** syntax: stateLoop

Such a stateLoop is useful when needing a successor for each element of an input set, for example for each node in the neighborhood of the current node.

```
forall{
        mouseTurn{
                counter++;
        }
        for(Graph.Node child: catPosition.getChildren()){
                mouseTurn{
                        counter++;
                        catPosition=child;
                }
        }
}
```

**Listing 3.15:** example for a forall block in alternating programs

### 3.1.5 Header

As we deal with Java Classes and our compiler generates Java Classes there are some Java Features for the Header. The first one is the package. At the beginning of an alternating program there is the definition of the Java package the compiled program should be included. The second feature in the header is the possibility of Java class imports. So we can use Java Classes in our alternating program without qualifying the whole package.

```
'package' qualifiedName ';'
'import' 'static'? qualifiedName ('.' '*')? ';'
```

**Listing 3.16:** syntax: header

```
package at.ac.tuwien.dbai.staff.dvorak.examples;
import java.util.Set;
import java.util.List;
```

**Listing 3.17:** example for a header in an alternating program

## 3.2 Programs

In the last section we introduced a language for writing alternating programs, but it would be hard to capture the beauty of this language without programs. In this section we present some alternating algorithms written in our language. First there are programs for some problems from Section 2.6. We then present a program simulating an alternating Turing machine. Then there are alternating programs for P-complete problems, which at first sight do not seem to be alternating problems. We close this section with an alternating program for the PSPACE-complete problem Tic-Tac-Toe.

### 3.2.1 Alternating Programs

Here we present some nice alternating programs for the intuitive alternating problems from Section 2.6. Further we try to give an idea how to write an alternating program starting with an alternating description of a problem.

#### Definite Horn Minimal Model

Here we give an alternating program for the definite horn minimal model problem we presented in Section 2.6.1. First we need a Java encoding of the problem. We decide to encode variables as characters and therefore we encode a problem instance as a triple:

`char goal`: the variable we test to be in the minimal model

`Map<Character, Set<Set<Character>>> rules`: a mapping from a head $h$ to the bodies of the rules with head $h$

`<Character> fact`: a set of facts.

We mentioned in the alternating approach 2.6.4 that a variable $p$ is in the minimal model of a definite Horn formula if it is either a fact or there is a rule with head $p$ and a body which is true in the minimal model. So we get two states in the program, the state head that tests if a variable is in the minimal model and the state body that tests if the body of a rule is in the minimal model. A configuration is characterized either by a positive literal or by a body we have to prove. So the program has two work variables `head, body` and further there is a counter used to prevent derivation cycles.

**Program:** Definite Horn Minimal Model

```
package at.ac.tuwien.dbai.staff.dvorak.alternation.examples;
import java.util.Map;
import java.util.Set;

atm Horn(char goal, Map<Character, Set<Set<Character>>> rules, Set<Character>
         facts) {
    char head=goal;
    Set<Character> body;
    int counter=rules.keySet().size();

    state head {
        // catch cycles
        if (counter<0) {
            reject;
        } else {
            counter--;
        }

        if (facts.contains(head)) {
            accept;
        } else {
            exists {
                for (Set<Character> rule: rules.get(head)) {
                    body {
                        body=rule;
                    }
                }
            }
        }
    }

    state body {
        forall{
            for (Character literal: body) {
                head {
                    head=literal;
                }
            }
        }
    }
}
```

---

As our framework has a technique to prevent cycles we can drop the counter in this program. Further it would make the tabled evaluation more efficient if we set body to the null reference for the head state instances and the head to null for the body state instances.

### Graph Game

Here we present a program solving the Acyclic Geography Game we presented in Section 2.6.2. The program even handles some generalizations. So the input graph can include cycles and nodes can be marked as winning fields; the player who enters the field wins.

The encoding of a problem instance is:

`char goal`: the variable we test to be in the minimal model

`Graph.Node<Position> startNode`: the node where the token is placed at the beginning

`Graph<Position> gameBoard`: the graph the game is played on

To describe the current game configuration we just need a cursor to the node with the token, the program saves this in the work variable `Graph.Node<Position> position`.

In two player games there is a winning strategy for player 1 if there is a move for player 1 such that all possible moves for player 2 lead to a winning position for player 1. So the program has two states, one for each player. Such a state first checks if the token is at a winning node, then the other player who moved there wins. If there is no immediate winning position the program follows all possible moves, in this case all outgoing edges.

**Program:** Graph Game

```
package at.ac.tuwien.dbai.staff.dvorak.alternation.examples;
import at.ac.tuwien.dbai.staff.dvorak.alternation.examples.GraphGame.Position;

atm GGAtm (Graph.Node<Position> startNode, Graph<Position> gameBoard) {
    Graph.Node<Position> position=startNode;

    state ExistsState {
        if (position.getObject().winning) {
            reject;
        } else {
            exists {
                for (Graph.Node<Position> node : position.getChildren()) {
                    ForallState {
                        position=node;
                    }
                }
            }
        }
    }

    state ForallState {
        if (position.getObject().winning) {
            accept;
        } else {
            forall {
                for (Graph.Node<Position> node : position.getChildren()) {
                    ExistsState {
                        position=node;
                    }
                }
            }

        }
    }
}
```

### Cat and Mouse

Cat and Mouse is another program searching for a winning strategy in a game from Section 2.6.2. The Cat and Mouse program is very similar to the program for the geography game. One difference is that there are two tokens on the graph one for the mouse and one for the cat, so beside the game board saved as graph and goal `Graph<Integer> gameboard, Graph.Node <Integer> goal` we need two start positions `Graph.Node<Integer> catStart, Graph.Node< Integer> mouseStart` and two work variables `Graph.Node<Integer> catPosition, Graph. Node<Integer> mousePosition` holding the current game configuration.

Another difference between the two games is that the mouse/player1 only wins when it reaches the goal and not when the cat/player2 has no possible moves. Further the cat and the mouse always have the choice to pass moving and stay at their node.

**Program:** Cat and Mouse

```
package at.ac.tuwien.dbai.staff.dvorak.alternation.examples;

atm CatAndMouse(Graph<Integer> gameboard, Graph.Node<Integer> catStart, Graph.
                Node<Integer> mouseStart,Graph.Node<Integer> goal) {
    Graph.Node<Integer> catPosition=catStart;
    Graph.Node<Integer> mousePosition=mouseStart;
    int counter=0;

    state mouseTurn {
        if (mousePosition==catPosition) {
            reject;
        }
        if (counter>gameboard.size()*gameboard.size()) {
            reject;
        }
        exists{
            // mouse doesn't move
            catTurn{
                counter++;
            }
            // mouse move
            for (Graph.Node child: mousePosition.getChildren()) {
                catTurn {
                    counter++;
                    mousePosition=child;
                }
            }
        }
    }

    state catTurn {
        if (mousePosition==goal) {
            accept;
        }
        if (mousePosition==catPosition) {
            reject;
        }
        forall{
            // cat doesn't move
            mouseTurn{
                counter++;
            }
            // cat move
            for (Graph.Node child: catPosition.getChildren()) {
                if (child!=goal) {
                    mouseTurn {
                        counter++;
                        catPosition=child;
                    }
                }
            }

        }
    }
}
```

---

The program uses a counter to eliminate strategies where cat and mouse run in cycles, with the approach of cycle detection[1] in our framework this won't be necessary and even bad for the performance. We get another optimization approach by observing that if the mouse passes, this leads to a cycle in the strategy when the cat passes. As the cat choices are universally quantified we would always get such a cycle in a winning strategy and therefore there is a smaller one without passing. So it is dispensable to follow such a passing successor and we can remove this possibility for the mouse player.

### Pebble

As third game we present a program for the two-person pebble game from Section 2.6.2. As this is a two-player game, the program has two states, one for each player, that compute all the possible moves and recursively evaluate if they lead to a winning strategy. The pebble game is encoded as follows: A pebble is encoded as String and all pebbles in the game are saved in a set `Set<String> pebbles`. The rules are saved as list of length 3 and pooled in `Set<List<String>> rules`. The pebble that should be reached is encoded as `String goal` and further the initial configuration for the game is in `String[] startSet`.

The rank of the game is implicitly encoded in the initial configuration, it is the dimension of the startSet. A configuration of the pebble game is an array `String[] S` holding the

[1]We discuss the cycle detection in Section 3.3.3

current pebbles on the table. This program is a nice example for using arrays in alternating programming. Without arrays we would have to write a new program for each rank $k$.

**Program:** Pebble

```
package at.ac.tuwien.dbai.staff.dvorak.alternation.examples;
import java.util.Set;
import java.util.List;
import java.util.Arrays;

atm PEBBLE(Set<String> pebbles,Set<List<String>> rules,String goal, String[]
           startSet) {
    String[] S = startSet.clone();
    int counter=(int)Math.pow(pebbles.size(),startSet.length);

    state player1 {
        Arrays.sort(S);
        if (Arrays.binarySearch(S,goal)>=0) reject;
        if (counter<0) reject;

        exists{
            for (List<String> rule : rules) {
                if (Arrays.binarySearch(S,rule.get(0))>=0 && Arrays.binarySearch(
                    S,rule.get(1))>=0 && Arrays.binarySearch(S,rule.get(2))<0) {
                    player2 {
                        S[Arrays.binarySearch(S,rule.get(0))]=rule.get(2);
                        counter--;
                    }
                }
            }
        }
    }

    state player2 {
        Arrays.sort(S);
        if (Arrays.binarySearch(S,goal)>=0) accept;
        if (counter<0) reject;

        forall{
            for (List<String> rule : rules) {
                if (Arrays.binarySearch(S,rule.get(0))>=0 && Arrays.binarySearch(
                    S,rule.get(1))>=0 && Arrays.binarySearch(S,rule.get(2))<0) {
                    player1 {
                        S[Arrays.binarySearch(S,rule.get(0))]=rule.get(2);
                        counter--;
                    }
                }
            }
        }
    }
}
```

---

One thing we have to care about is that the two-person pebble game is in polynomial time, only for a fixed rank. So if we take a too large value for the rank the run time and space requirements get bad. Using the counter technique makes things worse as the initial value of the counter grows exponentially with the rank.

The pebble game is also interesting in the one player version. So one player tries to reach the goal pebble by applying the interchange rules and we want to know if there exists a series of moves such that the goal pebble is reached. This is equivalent to the question in n-person pebble games if it is possible that a player interchanges the goal pebble. Clearly this problem is in NL and in fact it is NL-complete Kasai et al., 1979. We can solve this with an "alternating" algorithm by modifying the program above. First we discard the player2 state and then we update the player1 state as it is in Listing 3.18.

```
state player1 {
    Arrays.sort(S);
    if (Arrays.binarySearch(S,goal)>=0) reject;
    if (counter<0) reject;

    exists{
        for(List<String> rule : rules){
            if(Arrays.binarySearch(S,rule.get(0))>=0 && Arrays.
                binarySearch(S,rule.get(1))>=0 && Arrays.binarySearch
                (S,rule.get(2))<0){
                 player1{
                     S[Arrays.binarySearch(S,rule.get(0))]=rule.get
                         (2);
                     counter--;
                 }
            }
        }
    }
}
```

**Listing 3.18:** the states in one player pebble program

### Circuit Value

The program solving the Monotone Circuit Value Problem from Section 2.6.3 is a program that works with a single state but still has a quantifier alternation in the computation. A problem instance consists of the circuit `Circuit circuit` with an id for each gate, a series of input values `boolean[] input` and the id of output gate `int outIndex`. A configuration of the program is described by the gate `Circuit.Gate<Circuit.GateTypes> gate` that is currently evaluated. The state `evaluate` chooses the quantifier to use according to the type of the gate. If it is an OR gate the program quantifies existentially over the predecessor gates and universally for AND gates.

**Program:** Monotone Circuit Value Problem

```
package at.ac.tuwien.dbai.staff.dvorak.alternation.examples;
/** Monotone Circuit Value Problem **/
atm MCVP(Circuit circuit,boolean[] input,int outIndex) {
    Circuit.Gate<Circuit.GateTypes>  gate=circuit.getOutGate(outIndex);

    state evaluate {
        switch (gate.type) {
        case INPUT:
            if (input[circuit.getInputIndex(gate)]==true) {
                accept;
            } else {
                reject;
            }
        case AND:
            forall {
                for (Circuit.Gate<Circuit.GateTypes> tempGate: gate.getIn()) {
                    evaluate {
                        gate=tempGate;
                    }
                }
            }
        case OR:
            exists {
                for (Circuit.Gate<Circuit.GateTypes> tempGate: gate.getIn()) {
                    evaluate {
                        gate=tempGate;
                    }
                }
            }
        case OUTPUT:
            exists {
                for (Circuit.Gate<Circuit.GateTypes> tempGate: gate.getIn()) {
                    evaluate {
                        gate=tempGate;
                    }
                }
            }
        default:
            reject;
        }
    }
}
```

Even if we never reach the `default: reject;` code, it is necessary for a valid program as each computation possible for the compiler must have an alternating return value.

### 3.2.2 An Alternating Turing Machine

As our language is an implementation of the alternating concept introduced by alternating Turing machines, we are interested whether the language has the whole power of alternating Turing machines. We show this by simulating an alternating Turing machine in our language.

We simulate a 2-string alternating Turing machine with an input and a work tape over the alphabet of Java characters, where the symbol 'b' is reserved for the blank symbol. We encode an alternating Turing machine as a tuple of:

`char[] inputtape`: the input

`TransitionFunction delta`: a nice encoding of the relation $\delta$. The Javadoc documentation of this class can be found in the Appendix B.1.6

`Map<Integer,Character> stateType`: a mapping from the states into the set $\{A, R, F, E\}$ representing $\{accept, reject, \wedge, \vee\}$

`int initialState`: The initial state

`int C`: an upper bound for the space requirements

The simulating program encodes a configuration of such a machine as

`char[] worktape`: the current worktape

`int inputCursor, int workCursor`: the cursor position

`int activeState`: the active state

For technical reasons we use an additional work variable `first` for some initializing work.

The alternating program has one state that evaluates a configuration. Depending on the labeling of the states it either accepts/rejects or computes successor configurations and combines them with the corresponding quantifier.

**Program:** Alternating Turing Machine

```
package at.ac.tuwien.dbai.staff.dvorak.alternation.examples;
import java.util.Set;
import java.util.Map;

atm AlternatingTuringMachine(char[] inputtape, TransitionFunction delta, Map<
                             Integer,Character> stateType, int initialState,int
                             C) {
    char[] worktape=new char[C];
    int inputCursor = 0;
    int workCursor = 0;
    int activeState=initialState;
    boolean first =true;

    state alternatingTuringMachine {
        if (first) {
            java.util.Arrays.fill(worktape, 'b');
            first=false;
        }
        if (stateType.get(activeState).equals('A')) {
            accept;
        }
        if (stateType.get(activeState).equals('R')) {
            reject;
        }
        if (stateType.get(activeState).equals('F')) {
            forall {
                for (TransitionFunction.Output outputTuple : delta.compute(
                     inputtape[inputCursor],worktape[workCursor],activeState)) {
                    alternatingTuringMachine {
                        worktape[workCursor]=outputTuple.workSymbol;
                        activeState =outputTuple.newState;
                        inputCursor+=outputTuple.inputCursor;
                        workCursor+=outputTuple.workCursor;
                    }
                }
            }
        }
        if (stateType.get(activeState).equals('E')) {
            exists {
                for (TransitionFunction.Output outputTuple : delta.compute(
                     inputtape[inputCursor],worktape[workCursor],activeState)) {
                    alternatingTuringMachine {
                        worktape[workCursor]=outputTuple.workSymbol;
                        activeState =outputTuple.newState;
                        inputCursor+=outputTuple.inputCursor;
                        workCursor+=outputTuple.workCursor;
                    }
                }
            }
        }
        reject;
    }
}
```

### 3.2.3 More Alternating Programs

There are problems in P which have intuitive deterministic polynomial time algorithms, but also have a nice alternating approach. In the following we present two examples for such the problems.

#### Lexicographically First Maximal Clique

Here we present a deterministic and an alternating algorithm for the graph problem of finding a lexicographically first maximal clique. So we start with a definition of this problem

**Lexicographically First Maximal Clique (LFMC)**

***Given:*** An undirected graph $G = (V, E)$ with an ordering on the vertices and a vertex $v$
***Problem:*** Is $v$ in the lexicographically first maximal clique?
A clique is a set of nodes with an edge between every two nodes. So a clique is set $C \subseteq V$ with the property $v_i, v_j \in C \rightarrow (v_i, v_j) \in E$. A maximal clique is a clique such that no more nodes can be added without destroying the clique property.

*Ref:* Cook, 1985

A deterministic algorithm would simply compute the lexicographically first maximal clique iterating over the nodes according to their order. A vertex is added to the clique if it is connected with all vertices already in the clique. When the algorithm arrives at $v$ it can decide if $v$ is in the clique or not.

The alternating approach starts with $v$ and uses a recursive characterization of being in a lexicographically first maximal clique.

**Lemma 3.2.1.** *A node $v$ is in the lexicographically first maximal clique $C$ iff for all vertices $v_i < v$ holds that if $(v, v_i) \notin E$ then $v_i \notin C$.*

*Proof.* $\Rightarrow$ : $v \in C$ implies $\forall v_i \in C : (v, v_i) \in E$. So there is no $v_i$ with $v_i \leq v$, $(v, v_i) \notin E$ and $v_i \in C$. Therefore the right hand side holds.
$\Leftarrow$ : Adding $v$ to a clique makes this clique lexicographically smaller than adding any series of $v_j > v$. So we can ignore nodes with higher index.
If the right hand side holds we know that every vertex $v_i \in C$ is connected with $v$, other wise we have a $v_i$ with $((v, v_i) \notin E)$ and $v_i \in C$ $\lightning$.
As $v$ is connected to all vertices in the clique we can add $v$ without destroying the clique property and therefore $v$ is in the lexicographically first maximal clique. □

The alternating algorithm uses Lemma 3.2.1 to decide the Lexicographically First Maximal Clique problem. First we have to mention the negated statement: A node $v$ is not in the lexicographically first maximal clique $C$ iff there exists a vertex $v' < v$ with $(v, v') \notin E$ and $v' \in C$. So in the alternating program there is (i) one state that tests if a vertex is in that clique using the lemma and, (ii) one state that tests if a vertex is not in the clique using the negated lemma.

**Program:** Lexicographically First Maximal Clique

```
package at.ac.tuwien.dbai.staff.dvorak.alternation.examples;
/** Lexicographically First Maximal Clique **/
atm LFMC(UndirectedGraph<Integer> graph, UndirectedGraph.Node<Integer> vertex)
{

    UndirectedGraph.Node<Integer> node=vertex;

    state inClique {
        if (node.getObject().equals(1)) {
            accept;
        }
        forall {
            for (int i=1;i<node.getObject();i++) {
                if (!graph.get(i).getNeighbours().contains(node)) {
                    notInClique {
                        node=graph.get(i);
                    }
                }
            }
        }
    }

    state notInClique {
        if (node.getObject().equals(1)) {
            reject;
        }
        exists {
            for (int i=1;i<node.getObject();i++) {
                if (!graph.get(i).getNeighbours().contains(node)) {
                    inClique {
                        node=graph.get(i);
                    }
                }
            }
        }
    }
}
```

---

In the states the program always checks if the current node is the first node in the ordering and if so it accepts or rejects. This is not necessary as the first node has no successors and therefore a universally quantified state always accepts and a existentially quantified state always rejects what is exactly what we need here.

### Cellular Automata

The second problem is the problem of evaluating a cellular automatons cell state.

**One Dimensional Cellular Automata (CA)**

***Given:*** A one-dimensional cellular automaton and a state $q$, a cell $c$, a time bound $t$ in unary encoding and an initial configuration. The cellular automaton consists of a bi-infinite lattice where each square has a value from a finite alphabet $\Sigma$ and a local transition rule $\Phi : \Sigma^{2r+1} \to \Sigma$. We call $r$ the range. The function $\Phi$ computes the state of a cell at time $t+1$ out of the states from itself and its r-left/right neighbors at time $t$.

***Problem:*** Is the cell $c$ in state $q$ at time t?
*Ref:* Greenlaw et al., 1995, Lindgren and Nordahl, 1990, Alvy Ray Smith, 1971, Albert and Culik, 1987

The deterministic algorithm starts at time t'=0 with the initial configuration. In each iteration the algorithm computes the content of the $(t - t') \cdot r$ right and left neighbors of $c$ and for $c$ itself using the function $\Phi$.

Once more we use a recursive approach for the alternating program.

**Lemma 3.2.2.** *The cell $c_i$ is in state $q$ at time $t$ ($c_i(t) = q$) iff there exists a rule $q_{-r}, \dots, q_0, \dots, q_r \to q$ with $q_j \in \Sigma$ and for all $q_j$ it holds that $c_{i+j}(t-1) = q_j$.*

*Proof.* $\Rightarrow$ The cellular automaton used a rule to compute $c_i(t)$ out of the values $c_{i-r}(t-1), \dots, c_{i+r}(t-1)$. So we have found a rule satisfying the right hand side.
$\Leftarrow$ As a cellular automaton is deterministic ($\Phi$ is a function) the computation uses the rule from the right hand side and computes $c_i(t) = q$ □

The alternating program uses the idea of Lemma 3.2.2 to compute the state. So there is one state that branches over the rules deriving $q$ and an other state that checks the conditions to apply a rule.

**Program:** One Dimensional Cellular Automata

```
package at.ac.tuwien.dbai.staff.dvorak.alternation.examples;
import java.util.List;
import java.util.Set;
import java.util.Map;

atm OneDimensionalCellularAutomata (char[] cells, int cell, char finalstate, int
                                    T,Byte r, Map<List<Character>,Character>
                                    rphi) {
    List<Character> parents;
    int t=T;
    int position=cell;
    char caState=finalstate;

    //look for rules deriving the state
    state ExistsState {
        if (t==0) {
            if (position>=0 && position<cells.length) {
                if (caState==cells[position]) {
                    accept;
                } else {
                    reject;
                }
            } else {
                if (caState=='b') {
                    accept;
                } else {
                    reject;
                }
            }
        }
        exists{
            for (Map.Entry<List<Character>,Character> hilf : rphi.entrySet()) {
                if (hilf.getValue().equals(caState)) {
                    ForallState {
                        parents=hilf.getKey();
                    }
                }
            }
        }
    }

    //check preconditions for the selected rule
    state ForallState {
        t--;
        forall {
            for (int i=0 ; i<=2*r; i++) {
                ExistsState {
                    position=position-r+i;
                    caState=parents.get(i);
                }
            }
        }
    }
}
```

The CA-problem is in polynomial time according to the values of $r$ and $t$ but not according to the size of an efficient (logarithmic) encoding of them. So this problem is only in the class $\mathsf{P}$ if we encode $r$ and $t$ in a unary way.

### 3.2.4 Tic-tac-toe or PSPACE in our Framework

Problems hard for the class PSPACE are known as inherent hard problems, but when dealing with small instances of such problems we often can solve them in reasonable time and space. Here we present an alternating program that decides if a tic-tac-toe configuration has a winning strategy.

Tic-tac-toe is a two-player game played on a $3 \times 3$ grid. The players alternate marking a cell with their symbol (cross or naught). A player wins if he get a horizontal, vertical or diagonal row with his symbols. Tic-tac-toe can be generalized as game played on an $m \times n$ board and the players need $k$ cells in a row. It is well known that this generalization of tic-tac-toe is PSPACE-complete Reisch, 1980, but tic-tac-toe itself is a very small instance and therefore we can use our framework to compute winning strategies.

The alternating algorithm is very simple. The game grid is encoded as one dimensional array. The indices 0-2 for the first, 3-5 for the second and 6-8 for the third row. The program uses an initial state that decides which player starts moving. Then there is a state for each player, which first computes if the other player already has three in a row. Then it branches over all possible moves, the empty cells in the grid.

**Program:** TicTacToe

```
package at.ac.tuwien.dbai.staff.dvorak.alternation.examples;

atm TicTacToe (byte[] game, boolean crossPlayersTurn) {
    byte[] paper = game.clone();

    state start {
        if (crossPlayersTurn) {
            forall {
                crossPlayer{
                }
            }
        } else {
            forall {
                noughtPlayer{
                }
            }
        }
    }

    state crossPlayer {
        // check if there are three noughts in a row
        if (paper[4]==2) {
```

```
        if ( paper[0]==2 & paper[8]==2) reject;
        if ( paper[1]==2 & paper[7]==2) reject;
        if ( paper[2]==2 & paper[6]==2) reject;
        if ( paper[3]==2 & paper[5]==2) reject;
    }
    if (paper[0]==2 & paper[1]==2 & paper[2]==2) reject;
    if (paper[0]==2 & paper[3]==2 & paper[6]==2) reject;
    if (paper[2]==2 & paper[5]==2 & paper[8]==2) reject;
    if (paper[6]==2 & paper[7]==2 & paper[8]==2) reject;

    exists{
        for (int i=0;i<9;i++) {
            if (paper[i]==0) {
                noughtPlayer {
                    paper[i]=1;
                }
            }
        }
    }
}

state noughtPlayer {
    // check if there are three crosses in a row
    if (paper[4]==1) {
        if ( paper[0]==1 & paper[8]==1) accept;
        if ( paper[1]==1 & paper[7]==1) accept;
        if ( paper[2]==1 & paper[6]==1) accept;
        if ( paper[3]==1 & paper[5]==1) accept;
    }
    if (paper[0]==1 & paper[1]==1 & paper[2]==1) accept;
    if (paper[0]==1 & paper[3]==1 & paper[6]==1) accept;
    if (paper[2]==1 & paper[5]==1 & paper[8]==1) accept;
    if (paper[6]==1 & paper[7]==1 & paper[8]==1) accept;

    if (paper[0]!=0 & paper[1]!=0 & paper[2]!=0 & paper[3]!=0 & paper[4]!=0
        & paper[5]!=0 & paper[6]!=0 & paper[7]!=0 & paper[8]!=0) {
        reject;
    }
    forall {
        for (int i=0;i<9;i++) {
                if (paper[i]==0) {
                    crossPlayer {
                        paper[i]=2;
                    }
                }
            }
        }
    }
}
```

We computed the output for the empty game board on the authors computer in 4,83 milliseconds which is indeed feasible. (Intel(R) Core(TM)2 Duo CPU E7200 @ 2.53GHz) As the reader may know there is no winning strategy for tic-tac-toe.

## 3.3 The Framework

As we have a language for alternating programs it would be reasonable to have a framework for executing such programs. In this section we present a framework that efficiently simulates the alternating programs using the simulating technique from the proof of Theorem 2.5.4.

### 3.3.1 Used Technologies

First of all a list of technologies used in this framework:

- Sun Java 1.6 - http://java.sun.com/
- ANTLR 3.0.1 - http://www.antlr.org Parr, 2007
- StringTemplate 1.2 - http://www.stringtemplate.org/
- ANTLR Java 1.5 grammar Parr, 2008

### 3.3.2 Parts of the Framework

The framework consists of 3 parts:

- ANTLR Grammar for alternating programming language - see Appendix A
- A Compiler that translates an alternating programs into Java classes - see Appendix D
- A Java package for simulating alternating programs - see Appendix C

The grammar defines the accurate syntax for writing alternating programs. Based on this grammar the compiler is written in ANTLR and compiled into Java classes.

The compiler maps an alternating program to a single Java file but to multiple Java classes. The package *alternation.runtime* contains the Java classes for simulating the alternating algorithm. So this package has to be in the class path when executing an alternating program. This framework is available on the following website:

www.dbai.tuwien.ac.at/staff/dvorak/alternation/

### 3.3.3 Simulation Algorithms

In this section we present the idea of our simulation algorithms. The corresponding Java - classes are in the package *alternation.runtime* (Appendix C). The type for simulation algorithms is defined in the interface *alternation.runtime.ATM* (Appendix C.1.1).

We present three algorithms for simulation alternating programs. Starting with a very simple algorithm and improving it stepwise we present the standard algorithm in our framework. All three algorithms work in the following way.

1. initializing the work variables
2. Compute the leading-to-acceptance value of the initial state on these work variables.

The algorithms in this section use different ways to evaluate the the leading-to-acceptance value of a configuration of an alternating program. With a configuration of an alternating program we mean a pair of a state and a set of work variables instances.

#### Naive Simulation

The naive way to evaluate the leading-to-acceptance value of a configuration is:

1. Compute the deterministic code of the state until there is an alternating construct.
2. If this construct is an **accept** / **reject** statement, the algorithm returns **true** / **false**.
3. If this construct is a **forall** or **exists** construct the algorithm
   (a) computes all successor configurations and store them until there is a decision about the original configuration.
   (b) evaluates the leading-to-acceptance value of the successors and combines them, according to the quantifier, to a leading-to-acceptance value for the original configuration.
   We use a kind of short-circuit evaluation, so in the **forall** case the algorithm returns **false** when a successor is evaluated to **false** and returns **true** when all successors are evaluated to **true**. Similar holds for the **exists** case.

This algorithm makes something like a depth first search in the computation tree. So the worst case runtime is in proportion to the size of the computation tree. The computation tree in general grows exponentially with its input, as in worst case it branches in each step and uses polynomial time. So this is not a good algorithm for problems in the complexity class ALOGSPACE as we know from Theorem 2.5.5 that such problems can be solved in polynomial time. Further if there is a loop in the computation tree the algorithm may never halt. We can avoid this problem by using the counter technique from the proof of Theorem 2.5.1 but this shifts the responsibility to the programmer and, furthermore, is not optimal for running time and space behavior.

This algorithm is implemented in the Java class alternation.runtime.ATMsimple (Appendix C.2.3).

### Tabled evaluation

Following the idea from Theorem 2.2 we improve the naive algorithm. The idea is to add a table where we save all computed configurations with their leading-to-acceptance value.
The leading-to-acceptance value of a configuration is computed as follows:

1. Check if the configuration is in the table and if so return the value from the table
2. Save a copy of the configuration.
3. Compute the deterministic code of the state until there is an alternating construct.
4. If this construct is an **accept** / **reject** statement the algorithm returns **true** / **false**.
5. If this construct is a **forall** or **exists** construct the algorithm
   (a) computes all successor configurations and stores them until there is a decision about the original configuration.
   (b) evaluates the leading to acceptance value of the successors and combines them, according to the quantifier, to a leading to acceptance value for original configuration.
6. Store the copy of the original configuration and the acceptance value in the table.

The table prevents us from computing the same configuration multiple times. The runtime of this algorithm is in proportion to the number of configurations. For problems in the complexity class ALOGSPACE the number of configurations is bounded by a polynomial and therefore this algorithm is faster than the naive one. This algorithm saves all computed configurations so the required space is more or less in the order of the running time. This often leads to high space requirements.

The remaining problem is to handle the cycles in the computation tree. To prevent cycles, the programmer still has to use a counter or similar techniques. Such counters are usually initialized by a polynomial in the input. For the Horn minimal model it is simply the linear polynomial $n$. For Cat and Mouse the counter is $n^2$ and it gets worse for a pebble game with a start set of size 5 where the counter is initialized by $n^5$. We can encode such counters in $\mathsf{SPACE}(\log n)$ and so the counter has only a little effect to the size of configurations but the pebble counter gives a factor $n^5$ to the number of possible configurations and therefore to the running time and to the space requirements. In the worst case a problem configuration is evaluated once for each counter value.

This algorithm is implemented in the Java class alternation.runtime.ATMacyclic (Appendix C.2.1).

### Cycle detection

As we have seen the counter technique, a very useful concept for theoretic purposes, makes things bad in practical computation. But we are favored by fortune and so there is a simple solution for this problem. While the theoretic approach solves the problem at the level of alternating machines our approach handles cycles in the simulation algorithm instead of the alternating program.

We start with thinking about what a simulation algorithm should do when detecting a cycle. The answer is that the algorithm should reject the last edge of this computation path, because this cycle would never lead to a minimal solution. With rejecting an edge, we mean to send a reject to the successor's evaluation but don't save the reject value for the last node in the table.

This leads to an algorithm for handling cycles. The algorithm saves the current computation path in a list with some stack functionality. Whenever a computation revisits a configuration saved in this list, the algorithm simple determines this edge as false, what is exactly what we need for this computation path.

When using tabled evaluation we have to be a bit careful with this cycle detection technique. With cycle detection the evaluation of a configuration is no more independent of the computation path. An edge that leads to a cycle in one computation path may be a valid choice in another path. We resolve this problem by flagging rejects caused by cycles. As soon as the first node of the cycle, we call it cycle leader, is decided to an accept or non-flagged reject, the algorithm recomputes the leading-to-acceptance values of the states mapped to flagged rejects. If the first cycle leader in the computation path is determined to a flagged reject we save this as an ordinary reject.

So this algorithm works as follows:

1. Check if the configuration is in the table and if so return the value from the table

2. Check if the configuration is in the predecessor list and if so return a flaged reject.

3. Save a copy of the configuration.

4. Compute the deterministic code of the state until there is an alternating construct.

5. If this construct is an **accept** / **reject** statement the algorithm returns **true** / **false**.
6. If this construct is a **forall** or **exists** construct the algorithm
   (a) stores the copy of the original configuration and the acceptance value **false** in the table;
   (b) computes all successor configurations and stores them until we made a decision about the original configuration;
   (c) evaluates the leading-to-acceptance value of the successors and combines them, according to the quantifier, to a leading-to-acceptance value for original configuration.
7. If this configuration is a cycle leader then update the table according to the acceptance value.

This is the algorithm used by default in our framework, it is implemented in the Java class alternation.runtime.ATMcyclic (Appendix C.2.2).

### 3.3.4 The Compiler

As we have mentioned, the compiler maps an alternating program to several Java classes but only generates one file. So we handle nested classes. The main class adopts the alternating programs name and has the following nested classes:

- a class Inputtape which is a container for the input variables of the alternating program
- a container class Worktape to handle the work variables.
- a class for each state in the alternating algorithm. Such a class inherits a method to set the input of the computation and defines a method given a configuration of work variables computing the Result of the state. Such a Result is either accept/reject or a set of successors with a quantifier.

Listing 3.19 shows an example of such a Java file with nested classes.

```
public class Horn {
    private ATM<Inputtape, Worktape> atm;
    public class Inputtape {
        [...]
    }
    public class Worktape implements InterfaceWorktape<Inputtape> {
        [...]
    }
    private class head extends State<Inputtape, Worktape>  {
        public ResultTuple<Worktape> compute(Worktape atmWorktape) {
            [...]
        }
    }
    private class body State<Inputtape,Worktape>  {
        public ResultTuple<Worktape> compute(Worktape atmWorktape) {
            [...]
        }
    }
```

**Listing 3.19:** Classes in the compiled Horn program (simplified)

The main class itself has two methods. The compute method computes the alternating algorithm on an input. The method getComputationTree() method is the interface to the computation tree of the alternating computation.

```
    public boolean compute(char goal,Map<Character, Set<Set<
       Character>>> rules,Set<Character> facts) {
        [...]
    }
    public ComputationTree<Worktape> getComputationTree(){
        [...]
    }
}
```

**Listing 3.20:** Methods in the compiled Horn program (simplified)

### 3.3.5 The Computation Tree in our Framework

As we mentioned in Section 2.7 the computation tree is our resource for additional information about the solution. In our algorithm with tabled evaluation we save the whole computation tree and therefore we have access to this information.

After computing the decision of one problem instance we can access the computation tree with the getComputationTree() method. This method returns an object of the type alternation .runtime.ComputationTree (Appendix C.2.6), which offers a tree functionality to navigate between configurations and access to their work variables. So we can write programs playing winning strategies in a specific game or get a proof tree for a Horn problem

In Listing 3.21 there is an example program that extracts all rules from the proof tree of a definite Horn minimal model instance. We simply do this by recursive access to the leading to acceptance successors starting with the initial configuration, the root of the computation tree.

```
public static void drawProofTree(ComputationTree<Horn.Worktape> cT){
    drawNode(cT,cT.getRoot());
}
```

```
public static void drawNode(ComputationTree<Horn.Worktape> cT,
   ComputationNode<Horn.Worktape> node){
    List<ComputationNode<Horn.Worktape>> children=cT.getChildren(
        node);

    if (children!=null){
         if (node.isAllQuantified()){
              System.out.println(node.getWorktape().head + ":-␣"+ node
                  .getWorktape().body);
              for (ComputationNode<Horn.Worktape> child : children){
                   if (cT.get(child).equals(true)){
                        drawNode(cT,child);
                   }
              }
         } else {
              for (ComputationNode<Horn.Worktape> child : children){
                   drawNode(cT,child);
              }
         }
    }
}
```

**Listing 3.21:** Program that prints all clauses used to decide a definite Horn minimal model instance

# Chapter 4

# Experimental results

In this chapter we present some empirical tests with our framework. First we compare the running time behavior of alternating programs in contrast to deterministic programs for the same problems. In the second part of this chapter we compare the different simulation algorithms for alternating programs we introduced in Section 3.3.3.

The run time measurement data present in this chapter was generated on the following system.

**Test System**

- CPU: PE T300 Quad Core Xeon X3323, 2.5GHz, 2x3MB, 1333MHz FSB
- RAM: 4GB DDR2 667MHz Memory
- OS: SUSE Linux Enterprise Server 10
- Java: Java(TM) SE Runtime Environment (build 1.6.0 10-rc-b28)

Further we started the Java virtual machine with the following parameters *-Xms512m -Xmx1536m -Xss2024k*. So the programs where restricted to 1,5 GB memory.

## 4.1 Alternating Programs vs Iterative Programs

In this section we compare a deterministic program deciding the Horn minimal model problem with our alternating program. We are interested in their run time behavior and for the Horn program if the given memory suffices for a computation.

The deterministic algorithm computes the minimal model as follows. The set $M$ is initialized as empty set. Then the set iterates over the clauses of the formula. If all negated literals of a clause are already in $M$, then the positive literal is added to $M$ and the rule is removed. The algorithm accepts as soon we add the variable we ate interested in to $M$ and rejects if it iterates over all remaining rules without changing $M$. The Java implementation of this algorithm can be found in Listing 4.1.

```
public static boolean solveHorn(Map<Set<Integer>,Set<Integer>>
    clauses, int goal){
     Set<Integer> model=new HashSet<Integer>();
     if(clauses.containsKey(new HashSet<Integer>())){
          model.addAll(clauses.get(new HashSet<Integer>()));
     }
     boolean modify=true;
     while(modify){
          modify=false;
          for(Set<Integer> body :clauses.keySet() ){
               if (model.containsAll(body)){
                    Set<Integer> head=clauses.get(body);
                    if(head.contains(goal)){return true;}
                    model.addAll(clauses.get(body));
                    modify=true;
                    clauses.remove(body);
                    break;
               }

          }
     }
     return false;
}
```

**Listing 4.1:** Java program for the definite Horn minimal model problem

### 4.1.1 Test Data

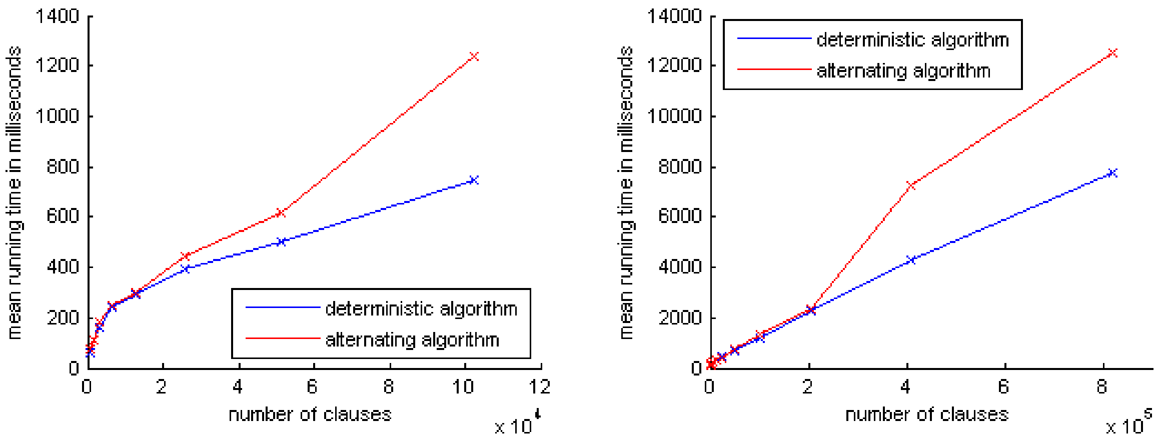
All starts with generating some examples to test the algorithms. A problem instance is a definite horn formula over some variables $\Sigma$ and a variable that should be tested. So we fix the set of variables to $\Sigma_1 = \{0, 1, \ldots, 48, 49\}$ or $\Sigma_2 = \{0, 1, \ldots, 98, 99\}$ and the goal to 0. To complete the instances we randomly generate a definite Horn formula as described in the following.

First we restrict the clauses of our horn formula to have at most three literals. Then we generate $k$ clauses by selecting variables uniformly distributed. In this process we eliminate duplicates such that we have $k$ different clauses in the formula. We generated test instances with different combinations of $\Sigma_i$ and $k$ over all 2700 instances.

| $\Sigma$ | k | # instances | $\Sigma$ | k | # instances |
|---|---|---|---|---|---|
| $\Sigma_1$ | 800 | 150 | $\Sigma_2$ | 1600 | 150 |
| $\Sigma_1$ | 1600 | 150 | $\Sigma_2$ | 3200 | 150 |
| $\Sigma_1$ | 3200 | 150 | $\Sigma_2$ | 6400 | 150 |
| $\Sigma_1$ | 6400 | 150 | $\Sigma_2$ | 12800 | 150 |
| $\Sigma_1$ | 12800 | 150 | $\Sigma_2$ | 25600 | 150 |
| $\Sigma_1$ | 25600 | 150 | $\Sigma_2$ | 51200 | 150 |
| $\Sigma_1$ | 51200 | 150 | $\Sigma_2$ | 102400 | 150 |
| $\Sigma_1$ | 102400 | 150 | $\Sigma_2$ | 204800 | 150 |
| | | | $\Sigma_2$ | 409600 | 150 |
| | | | $\Sigma_2$ | 819200 | 150 |

### 4.1.2 Results

As we see in Figure 4.1 the algorithms plays in the same league for a low density of clauses, while in average the deterministic algorithm is a little bit faster. For a high density of clauses which causes a very high branching in the alternating algorithm the deterministic algorithm is about two times faster.



(a) $\Sigma_1$ (b) $\Sigma_2$

**Figure 4.1:** deterministic Horn vs. alternating Horn program, mean running time according to the number of clauses and different number of variables

Overall the running time of alternating program is very near to the running time of the deterministic program for an average problem instance. The mean difference is about 480 milliseconds.

## 4.2 Simulation algorithms

In this section we compare the simple tabled evaluation algorithm with the algorithm improved by the cycle detection. For this purpose we used the Horn program and a slightly optimized version of the Cat and Mouse program from Section 3.2.1. In both programs we removed the stay possibility for the mouse, which never leads to a "minimal" winning strategy. In the algorithm for the cycle detection simulation we further removed the counter.

```
state mouseTurn {
    if (mousePosition==catPosition){
        reject;
    }
    exists{
        for(Graph.Node child: mousePosition.getChildren()){
            catTurn{
                mousePosition=child;
            }
        }
    }
}
```

**Listing 4.2:** mouse state in the algorithm with cycle detection

### 4.2.1 Test Data

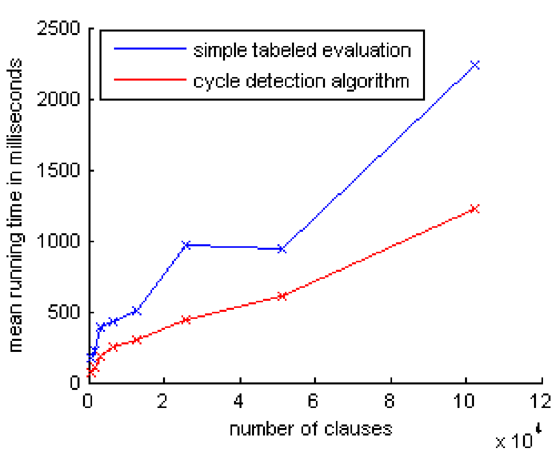
For the Horn algorithms we used test data described in section 4.1.1. The CatAndMouse algorithms required new test data. An instance of a cat and mouse game is a graph with three designated nodes. We fixed the set of nodes as $\{1, 2, 3, \dots, n\}$. The mouse starts at node 1, the cat at node 2 and the goal is 3. We generated random graphs by iterating over all possible edges and add them with probability $p$ in the graph. We vary $n$ in the values $\{10, 15, \dots, 45, 50\}$ and $p$ in $\{0.3, 0.5, 0.7\}$. For each combination of $n$ and $p$ we created 150 graphs and overall we get 4050 test instances.

### 4.2.2 Results

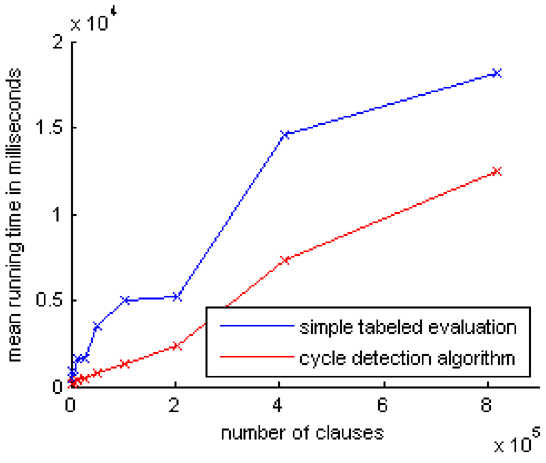
First we look at Horn problems where the value of the counter is only linear so that should not make things too worse. Figure 4.2 shows that even for this good case the algorithm with cycle detection is significantly faster.

Figure 4.3 shows why a good cycle detection is so important for our framework. The simple tabled evaluation algorithm is about a factor 10 slower than the improved one for graphs of size 50.

The reader may ask why there is no comparison with the simple simulation algorithm without tabled evaluation. The answer is that this algorithm is too bad, it can take days to get an answer for an instance even for graphs of size 10.

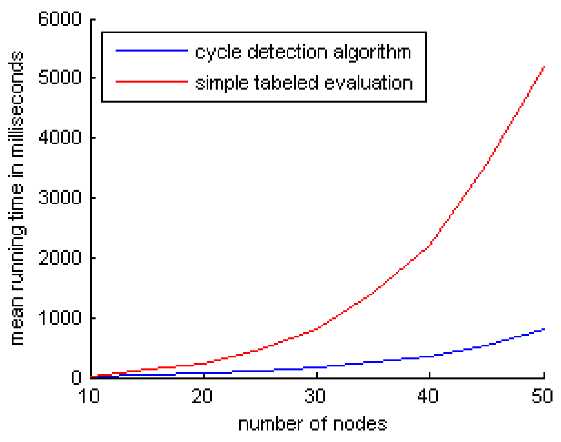


(a) $\Sigma_1$

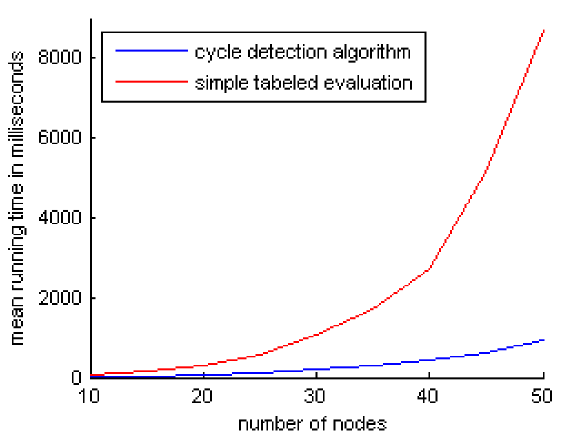


(b) $\Sigma_2$

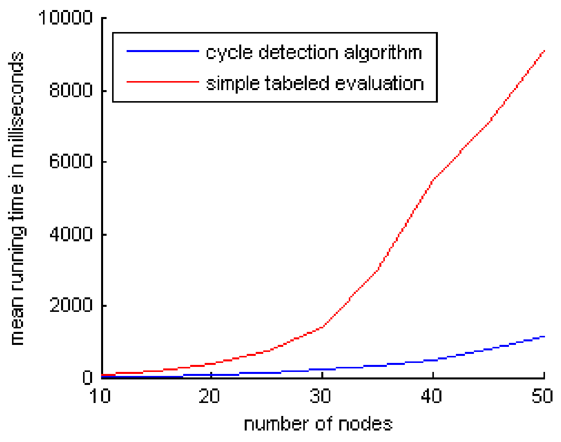
**Figure 4.2:** simple tabled evaluation vs. tabled evaluation with cycle detection , Horn programs mean running time according to the number of clauses and different number of variables



(a) p=0.3



(b) p=0.5



(c) p=0.7

**Figure 4.3:** simple tabled evaluation vs. tabled evaluation with cycle detection, Cat and Mouse programs mean running time according to the number of nodes and the edge density

# Chapter 5

# Summary & Conclusion

We explained why alternation is an important and useful programming concept in computational complexity theory and why it is desirable to have this concept available in practice. Further we identified the class ALOGSPACE as the class of feasible alternating problems.

In Section 2.6 we presented a range of natural alternating problems, i.e. problems that have a very intuitive alternating algorithm among them logic problems on Horn formulas, two player games, circuit evaluation problems and graph problems. We mentioned that these problems are feasible computable problems and therefore we have got a range of problems for which our framework is a benefit.

The language we introduced for alternating programming gives us a very intuitive tool to solve these problems. Further this languages also saves the power of Java and so that we can also use object-oriented features. The fact that our language is the appropriate method for alternating problems is confirmed by the program examples for these problems in Section 3.2.

Additional to solve the decision problems with alternating programs we also presented the computation tree as resource for additional information about the solution of the decision problem. So our framework has an interface to access the computation tree to construct for example a proof tree or a wining strategy.

The practical tests which we performed on our framework empirically showed that the running time of alternation programs simulated in our framework is almost as good as for deterministic programs. This means that the simulation algorithm in the framework does good work.

We have natural problems where alternating programming is the appropriate approach, a language to write alternating programs in a very intuitive way and a framework that executes the program in an efficient way. Therefore alternation is a programming paradigm that is useful for practical programming. Further the alternating paradigm is not restricted to decision problems, with the access to the computation tree we can do even more. In particular, having computed a winning strategy, we can use the computation tree to output or play this strategy.

## 5.1 Future Work

With our framework we developed a useful prototype for alternating programming, but there may be optimization potential in the simulation algorithm. So it seems to be wasted time to compute all successors in the cases where the algorithm only evaluates a few of them. A technique that computes the successors on demand when they are needed may be an improvement.

Another effect we observed is the fact that the order in which the successors are evaluated has a big effect on the running time. Thinking of games, it is bad if the move which leads immediately to a result is computed as last and the moves bringing no decision but causes big computation trees computed first. In many games this effect would be significantly reduced when adding some game specific heuristics for the evaluation order. So it would be a nice improvement for the language to add constructs that allows a programmer to modify the order of the successors.

# Appendix

# APPENDIX

This appendix offers additional resources for the alternation framework we introduced in the thesis.

- Appendix A: A ANTLR Grammar for the alternating programming language. It can be used to verify whether an alternating program fulfills the syntax of the language.
- Appendix B: The Javadoc documentation of the alternation.example package. Here we present the classes we used in our examples in all details.
- Appendix C: The Javadoc documentation of the alternation.runtime package. This is the package with the classes used for the different simulation algorithms.
- Appendix D: The Javadoc documentation of the main compiler class of our framework.

# Appendix A

# Grammar

```
grammar alternating_machine;

atm_file:
    packageDecl
    importDeclaration*
    head
        '{'
            worktape
            state+
        '}'
    ;

packageDecl:    'package' qualifiedName ';'
    ;

importDeclaration:   'import' 'static'? qualifiedName ('.' '*')? ';'
    ;

head    : (Identifier)? 'atm' Identifier  '('input')'
    ;

input   : arg(',' arg)*
    ;

arg : type Identifier
    ;

worktape: declaration*
    ;

declaration :
      args ';'
    | primitiveType Identifier  '=' expression';'
    | classOrInterfaceType Identifier  '=' expression';'
    | oneDimensionalArrayType Identifier '=' ( 'new' arrayCreator ';'
                                             |expression';')
    ;

oneDimensionalArrayType : (primitiveType| classOrInterfaceType)'['']'
    ;

args    : classOrInterfaceType Identifier (',' =Identifier)*
    ;

state   : 'state' Identifier '{' atmJava '}'
    ;

atmJava : statement+
    ;
```

```
typeList:   type (',' type)*
    ;

variableDeclarators:   variableDeclarator (',' variableDeclarator)*
```

```
    ;

variableDeclarator:   variableDeclaratorId ('=' variableInitializer)?
    ;

variableDeclaratorId:   Identifier ('[' ']')*
    ;

variableInitializer:
        arrayInitializer
    |   expression
    ;

arrayInitializer:   '{' (variableInitializer (',' variableInitializer)* (',')? )
                        ? '}'
    ;


type:
        classOrInterfaceType ('[' ']')*
    |   primitiveType ('[' ']')*
    ;


classOrInterfaceType:   Identifier typeArguments? ('.' Identifier typeArguments?
                                                   )*
    ;

primitiveType:
        'boolean'
    |   'char'
    |   'byte'
    |   'short'
    |   'int'
    |   'long'
    |   'float'
    |   'double'
    ;


typeArguments:   '<' typeArgument (',' typeArgument)* '>'
    ;

typeArgument:
        type
    |   '?' (('extends' | 'super') type)?
    ;
```

```
qualifiedName:   Identifier('.' Identifier)*
    ;

literal:
        integerLiteral
    |   FloatingPointLiteral
    |   CharacterLiteral
    |   StringLiteral
    |   booleanLiteral
    |   'null'
    ;

integerLiteral:
        HexLiteral
    |   OctalLiteral
    |   DecimalLiteral
    ;

booleanLiteral:
        'true'
    |   'false'
    ;

// STATEMENTS / BLOCKS

block:   '{' stmts+=blockStatement* '}'
    ;

blockStatement: statement
    ;

localVariableDeclaration:   type variableDeclarators
    ;


statement:
        block
    |   'if' parExpression statement ('else' statement)?
    |   'for' '(' forControl ')' statement
    |   'while' parExpression statement
    |   'do' statement 'while' parExpression ';'
    |   'try' block
        ( catches 'finally' block
        | catches
        |  'finally' block
        )
    |   'switch' parExpression '{' switchBlockStatementGroups '}'
    |   'throw' expression ';'
    |   'break' (Identifier)? ';'
    |   'continue' (Identifier)?  ';'
    |   ';'
    |   statementExpression ';'
    |   Identifier ':' statement
    |   ACCEPT ';'
    |   REJECT ';'
    |   FORALL '{' nondeterminism '}'
    |   EXISTS '{' nondeterminism '}'
    ;

catches:   (catchClause)+
    ;

catchClause:   'catch' '(' formalParameter ')' block
    ;

formalParameter:  type variableDeclaratorId
    ;

switchBlockStatementGroups:   (switchBlockStatementGroup)*
    ;

/* The change here (switchLabel -> switchLabel+) technically makes this grammar
   ambiguous; but with appropriately greedy parsing it yields the most
   appropriate AST, one in which each group, except possibly the last one, has
   labels and statements. */
switchBlockStatementGroup:  switchLabel+ blockStatement*
    ;

switchLabel:
        'case' constantExpression ':'
    |   'default' ':'
    ;

forControl:
        enhancedForControl
    |   forInit? ';' expression? ';' forUpdate?
    ;

forInit:
        localVariableDeclaration
    |   expressionList
    ;

enhancedForControl:   type Identifier ':' expression
    ;
```

```
forUpdate:   expressionList
    ;

// EXPRESSIONS

parExpression:   '(' expression ')'
    ;

expressionList:   expression (',' expression)*
    ;

statementExpression:   expression
    ;

constantExpression:   expression
    ;

expression:   conditionalExpression (assignmentOperator expression)?
    ;

assignmentOperator:
        '='
    |   '+='
    |   '-='
    |   '*='
    |   '/='
    |   '&='
    |   '|='
    |   '^='
    |   '%='
    |   ('<' '<' '=')=> t1='<' t2='<' t3='='
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() &&
          $t2.getLine() == $t3.getLine() &&
          $t2.getCharPositionInLine() + 1 == $t3.getCharPositionInLine() }?
    |   ('>' '>' '>' '=')=> t1='>' t2='>' t3='>' t4='='
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() &&
          $t2.getLine() == $t3.getLine() &&
          $t2.getCharPositionInLine() + 1 == $t3.getCharPositionInLine() &&
          $t3.getLine() == $t4.getLine() &&
          $t3.getCharPositionInLine() + 1 == $t4.getCharPositionInLine() }?
    |   ('>' '>' '=')=> t1='>' t2='>' t3='='
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() &&
          $t2.getLine() == $t3.getLine() &&
          $t2.getCharPositionInLine() + 1 == $t3.getCharPositionInLine() }?
    ;

conditionalExpression:   conditionalOrExpression ( '?' expression ':' expression
                                                    )?
    ;

conditionalOrExpression:   conditionalAndExpression ( '||'
                                                       conditionalAndExpression )
                                                       *
    ;

conditionalAndExpression:   inclusiveOrExpression ( '&&' inclusiveOrExpression )
                                                    *
    ;

inclusiveOrExpression:   exclusiveOrExpression ( '|' exclusiveOrExpression )*
    ;

exclusiveOrExpression:   andExpression ( '^' andExpression )*
    ;

andExpression:   equalityExpression ( '&' equalityExpression )*
    ;

equalityExpression:   instanceOfExpression ( ('==' | '!=') instanceOfExpression )
                                               *
    ;

instanceOfExpression:   relationalExpression ('instanceof' type)?
    ;

relationalExpression:   shiftExpression ( relationalOp shiftExpression )*
    ;

relationalOp:
        ('<' '=')=> t1='<' t2='='
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() }?
    |   ('>' '=')=> t1='>' t2='='
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() }?
    |   '<'
    |   '>'
    ;

shiftExpression:   additiveExpression ( shiftOp additiveExpression )*
    ;
```

```
shiftOp:
        ('<' '<')=> t1='<' t2='<'
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() }?
    |   ('>' '>' '>')=> t1='>' t2='>' t3='>'
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() &&
          $t2.getLine() == $t3.getLine() &&
          $t2.getCharPositionInLine() + 1 == $t3.getCharPositionInLine() }?
    |   ('>' '>')=> t1='>' t2='>'
        { $t1.getLine() == $t2.getLine() &&
          $t1.getCharPositionInLine() + 1 == $t2.getCharPositionInLine() }?
    ;


additiveExpression:   multiplicativeExpression ( ('+' | '-')
                                                      multiplicativeExpression )*
    ;

multiplicativeExpression:   unaryExpression ( ( '*' | '/' | '%' )
                                                   unaryExpression )*
    ;

unaryExpression:
       '+' unaryExpression
    |   '-' unaryExpression
    |   '++' unaryExpression
    |   '--' unaryExpression
    |   unaryExpressionNotPlusMinus
    ;

unaryExpressionNotPlusMinus:
        '~' unaryExpression
    |   '!' unaryExpression
    |   castExpression
    |   primary selector* ('++'|'--')?
    ;

castExpression:
       '(' primitiveType ')' unaryExpression
    |  '(' (type | expression) ')' unaryExpressionNotPlusMinus
    ;

primary:
        parExpression
    |   literal
    |   id=Identifier ('.' Identifier)* identifierSuffix?
    |   primitiveType ('[' ']')* '.' 'class'
    |   'void' '.' 'class'
    ;

identifierSuffix:
        ('[' ']')+ '.' 'class'
    |   ('[' expression ']')+ // can also be matched by selector, but do here
    |   arguments
    |   '.' 'class'
    |   '.' explicitGenericInvocation
    ;

arrayCreator:
        createdName '['
        (  ']' arrayInitializer
        |   expression ']'
        )
    ;

createdName:
        classOrInterfaceType
    |   primitiveType
    ;

explicitGenericInvocation:   nonWildcardTypeArguments Identifier arguments
    ;

nonWildcardTypeArguments:   '<' typeList '>'
    ;

selector:
        '.' Identifier arguments?
    |   '[' expression ']'
    ;

arguments:   '(' expressionList? ')'
    ;

// forall - exists
nondeterminism:
        (stateCall
    |   stateLoop)+
    ;

stateCall:
        Identifier stateBlock
    |   'if' parExpression '{' stateCall+ '}' ('else' '{'stateCall+'}')?
    ;
```

```
stateLoop:
        'for' '(' forControl ')' '{' stateCall '}'
    ;

stateBlock:   '{' stateStatement* '}'
    ;

stateStatement:
        stateBlock
    |   'if' parExpression stateStatement ('else' stateStatement)?
    |   'for' '(' forControl ')' stateStatement
    |   'while' parExpression stateStatement
    |   'do' stateStatement 'while' parExpression ';'
    |   'try' b1=stateBlock
        ( catches 'finally' stateBlock
        | catches
        |   'finally' stateBlock
        )
    |   'switch' parExpression '{' switchBlockStatementGroups '}'
    |   'throw' expression ';'
    |   'break' (Identifier)? ';'
    |   'continue' (Identifier)?  ';'
    |   ';'
    |   statementExpression ';'
    |   Identifier ':' statement
    ;
// LEXER
ACCEPT  :   'accept';

REJECT  :   'reject';

FORALL  :   'forall';

EXISTS  :   'exists';


HexLiteral : '0' ('x'|'X') HexDigit+ IntegerTypeSuffix? ;

DecimalLiteral : ('0' | '1'..'9' '0'..'9'*) IntegerTypeSuffix? ;

OctalLiteral : '0' ('0'..'7')+ IntegerTypeSuffix? ;

fragment
HexDigit : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
IntegerTypeSuffix : ('l'|'L') ;
```

```
FloatingPointLiteral:
        ('0'..'9')+ '.' ('0'..'9')* Exponent? FloatTypeSuffix?
    |   '.' ('0'..'9')+ Exponent? FloatTypeSuffix?
    |   ('0'..'9')+ Exponent FloatTypeSuffix?
    |   ('0'..'9')+ FloatTypeSuffix
    ;

fragment
Exponent : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

fragment
FloatTypeSuffix : ('f'|'F'|'d'|'D') ;

CharacterLiteral:   '\'' ( EscapeSequence | ~('\''|'\\') ) '\''
    ;

StringLiteral:  '"' ( EscapeSequence | ~('\\'|'"') )* '"'
    ;

fragment
EscapeSequence:
        '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UnicodeEscape
    |   OctalEscape
    ;

fragment
OctalEscape:
        '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UnicodeEscape:   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    ;

Identifier:   Letter (Letter|JavaIDDigit)*
    ;

fragment
Letter:
        '\u0024' |
        '\u0041'..'\u005a' |
        '\u005f' |
        '\u0061'..'\u007a' |
        '\u00c0'..'\u00d6' |
        '\u00d8'..'\u00f6' |
        '\u00f8'..'\u00ff' |
        '\u0100'..'\u1fff' |
        '\u3040'..'\u318f' |
        '\u3300'..'\u337f' |
        '\u3400'..'\u3d2d' |
        '\u4e00'..'\u9fff' |
        '\uf900'..'\ufaff'
    ;

fragment
JavaIDDigit:
        '\u0030'..'\u0039' |
        '\u0660'..'\u0669' |
        '\u06f0'..'\u06f9' |
        '\u0966'..'\u096f' |
        '\u09e6'..'\u09ef' |
        '\u0a66'..'\u0a6f' |
        '\u0ae6'..'\u0aef' |
        '\u0b66'..'\u0b6f' |
        '\u0be7'..'\u0bef' |
        '\u0c66'..'\u0c6f' |
        '\u0ce6'..'\u0cef' |
        '\u0d66'..'\u0d6f' |
        '\u0e50'..'\u0e59' |
        '\u0ed0'..'\u0ed9' |
        '\u1040'..'\u1049'
   ;

CMT: '/*'(options {greedy=false;}:.)* '*/'{$channel = HIDDEN;};
LCMT: '//' ~('\n'|'\r')* '\r'? '\n' {$channel = HIDDEN;};
WS:  (' '|'\t'|'\r'|'\n')+ {$channel = HIDDEN;};
```

# Appendix B

# alternation.examples

*Package Contents* *Page*

## B.1 Classes

### B.1.1 Class **Circuit**

A boolean circuit

DECLARATION

public class Circuit
**extends** java.lang.Object

CONSTRUCTORS

- *Circuit*
  `public` **Circuit**`( )`

METHODS

- *addGate*
  `public void` **addGate**`( int` **gateNumber,**
  `alternation.examples.Circuit.GateTypes` **type** `)`
  - **Usage**
    * add a gate to the circuit
  - **Parameters**
    * `gateNumber` - gate identifier
    * `type` - type of the gate

- *connect*
  `public void` **connect**`( alternation.examples.Circuit.Gate` **from,**
  `alternation.examples.Circuit.Gate` **to** `)`
  - **Usage**
    * connect a gates output with another gates input
  - **Parameters**
    * `from` -
    * `to` -

- *getGate*
  `public Circuit.Gate` **getGate**`( int` **gateNumber** `)`
  - **Parameters**
    * `gateNumber` - identifier
  - **Returns** - the gate with the specified identifier

- *getInputGate*
  `public Circuit.Gate` **getInputGate**`( int` **index** `)`

- **Parameters**
  - `index` -
- **Returns** - the specified input Gate

---

- *getInputIndex*
  `public int` **getInputIndex**`( alternation.examples.Circuit.Gate` **gate** `)`
  - **Parameters**
    - `gate` -
  - **Returns** - number in the ordering of input gates

---

- *getOutGate*
  `public Circuit.Gate` **getOutGate**`( int` **index** `)`
  - **Parameters**
    - `index` -
  - **Returns** - the specified output Gate

### B.1.2 Class **Circuit.Gate**

A Gate in the boolean circuit

#### Declaration

class Circuit.Gate
**extends** java.lang.Object

#### Fields

- public Object type
  -

#### Constructors

- *Circuit.Gate*
  `public` **Circuit.Gate**`( java.lang.Object` **type** `)`

#### Methods

- *addIn*
  `boolean` **addIn**`( alternation.examples.Circuit.Gate` **from** `)`

---

- *addOut*
  `boolean` **addOut**`( alternation.examples.Circuit.Gate` **to** `)`

---

- *getIn*
  `public Set` **getIn**`( )`

- *getOut*
  `public Set` **getOut**`( )`

### B.1.3 Class **Circuit.GateTypes**

The different types of gates

#### Declaration

static final class Circuit.GateTypes
**extends** java.lang.Enum

#### Fields

- public static final Circuit.GateTypes AND
  –
- public static final Circuit.GateTypes OR
  –
- public static final Circuit.GateTypes NOT
  –
- public static final Circuit.GateTypes INPUT
  –
- public static final Circuit.GateTypes OUTPUT
  –

#### Methods

- *valueOf*
  `public static Circuit.GateTypes` **valueOf**`( java.lang.String` **name** `)`

- *values*
  `public static Circuit.GateTypes` **values**`( )`

METHODS INHERITED FROM CLASS `java.lang.Enum`

- *clone*
  `protected final Object` **clone**`( )`
- *compareTo*
  `public final int` **compareTo**`( java.lang.Enum` **arg0** `)`
- *equals*
  `public final boolean` **equals**`( java.lang.Object` **arg0** `)`
- *finalize*
  `protected final void` **finalize**`( )`
- *getDeclaringClass*
  `public final Class` **getDeclaringClass**`( )`
- *hashCode*
  `public final int` **hashCode**`( )`
- *name*
  `public final String` **name**`( )`
- *ordinal*
  `public final int` **ordinal**`( )`
- *toString*
  `public String` **toString**`( )`
- *valueOf*
  `public static Enum` **valueOf**`( java.lang.Class` **arg0**`, java.lang.String` **arg1** `)`

## B.1.4 CLASS **Graph**

A directed graph

DECLARATION

public class Graph
**extends** java.lang.Object
**implements** java.lang.Iterable

CONSTRUCTORS

- *Graph*
  `public` **Graph**`( )`

METHODS

- *addEdge*
  `public boolean` **addEdge**`( java.lang.Object` **node1**`, java.lang.Object` **node2** `)`
- *addNode*
  `public boolean` **addNode**`( java.lang.Object` **node** `)`

- *get*
  `public Graph.Node` **get**`( java.lang.Object` **node** `)`
- *iterator*
  `public Iterator` **iterator**`( )`
- *removeEdge*
  `public boolean` **removeEdge**`( java.lang.Object` **node1**`, java.lang.Object` **node2** `)`
- *removeNode*
  `public boolean` **removeNode**`( java.lang.Object` **node** `)`
- *size*
  `public int` **size**`( )`
  - **Returns** - number of nodes

### B.1.5 Class **Graph.Node**

Node in a directed graph

#### Declaration

public static class Graph.Node
**extends** java.lang.Object

#### Constructors

- *Graph.Node*
  `public` **Graph.Node**`( java.lang.Object` **object** `)`

#### Methods

- *addChild*
  `public boolean` **addChild**`( alternation.examples.Graph.Node` **child** `)`
- *addParent*
  `public boolean` **addParent**`( alternation.examples.Graph.Node` **parent** `)`
- *childIterator*
  `public Iterator` **childIterator**`( )`
- *getChildren*
  `public Set` **getChildren**`( )`
- *getObject*
  `public Object` **getObject**`( )`
- *getParents*
  `public Set` **getParents**`( )`

- *isLinked*
  `public boolean` **isLinked**`( )`

---

- *parentIterator*
  `public Iterator` **parentIterator**`( )`

---

- *removeChild*
  `public boolean` **removeChild**`( alternation.examples.Graph.Node` **child** `)`

---

- *removeParent*
  `public boolean` **removeParent**`( alternation.examples.Graph.Node` **parent** `)`

---

- *toString*
  `public String` **toString**`( )`

### B.1.6 Class **TransitionFunction**

Transition Function delta for a nondeterministic 2-string Turing machines

#### Declaration

public class TransitionFunction
**extends** java.lang.Object

#### Constructors

- *TransitionFunction*
  `public` **TransitionFunction**`( )`

#### Methods

- *addRule*
  `public void` **addRule**`( char` **inputSymbol**`, char` **workSymbol**`, int` **state**`, char` **newWorkSymbol**`, int` **newState**`, int` **inputCursor**`, int` **workCursor** `)`
  - **Usage**
    * adds a transition rule
  - **Parameters**
    * `inputSymbol` -
    * `workSymbol` -
    * `state` -
    * `newWorkSymbol` -
    * `newState` -
    * `inputCursor` -
    * `workCursor` -

---

- *compute*
  `public Set` **compute**`( char` **inputSymbol**`, char` **workSymbol**`, int` **state** `)`

- **Usage**
  - computes the transition function
- **Parameters**
  - `inputSymbol` -
  - `workSymbol` -
  - `state` -
- **Returns** - the set of possible successors

### B.1.7 Class **TransitionFunction.Input**

An Input tuple (input symbol, worksymbol, state) for the transition function

#### Declaration

static class TransitionFunction.Input
**extends** java.lang.Object

#### Constructors

- *TransitionFunction.Input*
  `public` **TransitionFunction.Input(** `char` **inputSymbol,** `char` **workSymbol,** `int` **state )**

#### Methods

- *equals*
  `public boolean` **equals(** `java.lang.Object` **obj )**
- *hashCode*
  `public int` **hashCode( )**

### B.1.8 Class **TransitionFunction.Output**

An tuple in the Output Set (work symbol, new state, inputCursor movement, workCursor movement)

#### Declaration

static class TransitionFunction.Output
**extends** java.lang.Object

#### Constructors

- *TransitionFunction.Output*
  `public` **TransitionFunction.Output(** `char` **workSymbol,** `int` **newState,** `int` **inputCursor,** `int` **workCursor )**

## Methods

- *equals*
  `public boolean` **equals**`( java.lang.Object` **obj** `)`
- *hashCode*
  `public int` **hashCode**`( )`

### B.1.9 Class **UndirectedGraph**

An undirected graph

## Declaration

public class UndirectedGraph
**extends** java.lang.Object
**implements** java.lang.Iterable

## Constructors

- *UndirectedGraph*
  `public` **UndirectedGraph**`( )`

## Methods

- *addEdge*
  `public boolean` **addEdge**`( java.lang.Object` **node1**`, java.lang.Object` **node2** `)`
- *addNode*
  `public boolean` **addNode**`( java.lang.Object` **node** `)`
- *get*
  `public UndirectedGraph.Node` **get**`( java.lang.Object` **position** `)`
- *iterator*
  `public Iterator` **iterator**`( )`
- *removeEdge*
  `public boolean` **removeEdge**`( java.lang.Object` **node1**`, java.lang.Object` **node2** `)`
- *removeNode*
  `public boolean` **removeNode**`( java.lang.Object` **node** `)`
- *size*
  `public int` **size**`( )`

### B.1.10 Class **UndirectedGraph.Node**

A node in an undirected graph

#### Declaration

public static class UndirectedGraph.Node
**extends** java.lang.Object

#### Constructors

- *UndirectedGraph.Node*
  `public` **UndirectedGraph.Node**`(` `java.lang.Object` **object** `)`

#### Methods

- *addNeighbour*
  `public boolean` **addNeighbour**`(` `alternation.examples.UndirectedGraph.Node` **child** `)`
- *getNeighbours*
  `public Set` **getNeighbours**`( )`
- *getObject*
  `public Object` **getObject**`( )`
- *isLinked*
  `public boolean` **isLinked**`( )`
- *neighboursIterator*
  `public Iterator` **neighboursIterator**`( )`
- *removeNeighbour*
  `public boolean` **removeNeighbour**`(` `alternation.examples.UndirectedGraph.Node` **child** `)`
- *toString*
  `public String` **toString**`( )`

# Appendix C

# alternation.runtime

*Package Contents* *Page*

## C.1 Interfaces

### C.1.1 Interface **ATM**

simulation algorithm for alternating programs

#### Declaration

public interface ATM

#### Methods

- *compute*
  `public boolean` **`compute(`** `java.lang.Object` **input** `)`
  - **Usage**
    - simulates the machine on the given input
  - **Parameters**
    - `input` - Container with the input
  - **Returns** - true if accepts and false if rejects
- *getComputationTree*
  `public ComputationTree` **`getComputationTree(`** `)`
  - **Returns** - computation tree if available otherwise null

### C.1.2 Interface **InterfaceWorktape**

Container of work variables in our alternating machine.

#### Declaration

public interface InterfaceWorktape

#### Methods

- *clone*
  `public InterfaceWorktape` **`clone(`** `)`
  - **Returns** - a (flat) copy from the worktape
- *reset*
  `public void` **`reset(`** `java.lang.Object` **input** `)`
  - **Parameters**
    - `input` - initialize the worktape with the given input

## C.2 Classes

### C.2.1 Class **ATMacyclic**

This class uses tabled evaluation to simulate an alternating program.

#### Declaration

public class ATMacyclic
**extends** java.lang.Object
**implements** ATM

#### Constructors

- *ATMacyclic*
  `public` **ATMacyclic**`( java.util.Map` **states**`,`
  `alternation.runtime.InterfaceWorktape` **worktape**`, byte` **startState** `)`
  - **Usage**
    - constructor
  - **Parameters**
    - `states` - Map with states
    - `worktape` - a (empty) worktape
    - `startState` - stateid from the state the computation should start with

#### Methods

- *compute*
  `public boolean` **compute**`( java.lang.Object` **input** `)`
- *getComputationTree*
  `public ComputationTree` **getComputationTree**`( )`

### C.2.2 Class **ATMcyclic**

This class uses tabled evaluation and cycle detection to simulate an alternating program.

#### Declaration

public class ATMcyclic
**extends** java.lang.Object
**implements** ATM

CONSTRUCTORS

- *ATMcyclic*
  `public` **ATMcyclic**`( java.util.Map` **states**`,`
  `alternation.runtime.InterfaceWorktape` **worktape**`, byte` **startState** `)`
  - **Usage**
    - constructor
  - **Parameters**
    - `states` - Map with states
    - `worktape` - a (empty) worktape
    - `startState` - stateid from the state the computation should start with

METHODS

- *compute*
  `public boolean` **compute**`( java.lang.Object` **input** `)`
- *getComputationTree*
  `public ComputationTree` **getComputationTree**`( )`

## C.2.3 CLASS **ATMsimple**

This class simulates an alternating machine. (without tabled evaluation)

DECLARATION

public class ATMsimple
**extends** java.lang.Object
**implements** ATM

CONSTRUCTORS

- *ATMsimple*
  `public` **ATMsimple**`( java.util.Map` **states**`,`
  `alternation.runtime.InterfaceWorktape` **worktape**`, byte` **startState** `)`
  - **Parameters**
    - `states` - algorithms states
    - `worktape` - an empty worktape
    - `startState` - stateid from the state the computation should start with

METHODS

- *compute*
  `public boolean` **compute**`( java.lang.Object` **input** `)`
- *getComputationTree*
  `public ComputationTree` **getComputationTree**`( )`

### C.2.4 Class **ComputationNode**

Node in the ComputionTree holds: # the state of the machine # the worktape with the work variables # the Quantifier associed with the node - unset if halting ( immediate accepting / rejecting) node

#### Declaration

public class ComputationNode
**extends** java.lang.Object

#### Constructors

- *ComputationNode*
  `public` **ComputationNode**`( byte` **state**,
  `alternation.runtime.InterfaceWorktape` **worktape** `)`
  - **Parameters**
    - `state` -
    - `worktape` -

#### Methods

- *equals*
  `public boolean` **equals**`( java.lang.Object` **obj** `)`
- *getState*
  `public byte` **getState**`( )`
  - **Returns** - the id of the state
- *getWorktape*
  `public InterfaceWorktape` **getWorktape**`( )`
  - **Returns** - container with work variables
- *hashCode*
  `public int` **hashCode**`( )`
- *isAllQuantified*
  `public Boolean` **isAllQuantified**`( )`
  - **Returns** - true if universal quantified, false if existential quantified otherwise null
- *setQuantifier*
  `public void` **setQuantifier**`( boolean` **forallState** `)`
  - **Parameters**
    - `forallState` - true for a universal quantifier and false for a existential Quantifier
- *toString*
  `public String` **toString**`( )`

### C.2.5 Class **ComputationTree**

Computation Tree of an alternating computation - saves the configurations and their succesor relation

#### Declaration

public class ComputationTree
**extends** java.util.HashMap

#### Serializable Fields

- private Map stateIdToName
  –
- private Map nameToStateId
  –
- private ComputationNode root
  –

#### Constructors

- *ComputationTree*
  `public` **ComputationTree**`( )`

#### Methods

- *addEdge*
  `public void` **addEdge**`( alternation.runtime.ComputationNode` **parent**`, alternation.runtime.ComputationNode` **child** `)`
  - **Usage**
    - Adds a edge between two Computation Nodes
  - **Parameters**
    - `parent` - fromNode
    - `child` - toNode

---

- *getChildren*
  `public List` **getChildren**`( alternation.runtime.ComputationNode` **node** `)`
  - **Parameters**
    - `node` - ComputationNode
  - **Returns** - Successors in the ComputationTree

---

- *getRoot*
  `public ComputationNode` **getRoot**`( )`
  - **Returns** - the root of the computation tree

---

- *getStateId*
  `public Byte` **getStateId**`( java.lang.String` **name** `)`
  - **Usage**
    - returns a states id for a given name
  - **Parameters**
    - `name` - state name
  - **Returns** - state id

---

- *getStateName*
  `public String` **getStateName**`( java.lang.Byte` **stateId** `)`
  - **Usage**
    - returns a states name for a given id
  - **Parameters**
    - `stateId` - state id
  - **Returns** - state name

---

- *put*
  `public Boolean` **put**`( alternation.runtime.ComputationNode` **node**`, java.lang.Boolean` **acceptance** `)`

---

- *setNameToStateId*
  `public void` **setNameToStateId**`( java.lang.String` **string**`, java.lang.Byte` **b** `)`
  - **Usage**
    - maps the state name with its id
  - **Parameters**
    - `string` - state name
    - `b` - state id

---

- *setRoot*
  `public void` **setRoot**`( alternation.runtime.ComputationNode` **root** `)`
  - **Parameters**
    - `root` - ComputationNode constucted by the input and the initial worktape

---

- *setStateIdToName*
  `public void` **setStateIdToName**`( java.lang.Byte` **b**`, java.lang.String` **string** `)`
  - **Usage**
    - maps the state id with its name
  - **Parameters**
    - `b` - state id
    - `string` - state name

## METHODS INHERITED FROM CLASS `java.util.HashMap`

- *addEntry*
  `void` **addEntry**`( int` **arg0**`, java.lang.Object` **arg1**`, java.lang.Object` **arg2**`, int` **arg3** `)`
- *capacity*
  `int` **capacity**`( )`
- *clear*
  `public void` **clear**`( )`
- *clone*
  `public Object` **clone**`( )`
- *containsKey*
  `public boolean` **containsKey**`( java.lang.Object` **arg0** `)`
- *containsValue*
  `public boolean` **containsValue**`( java.lang.Object` **arg0** `)`
- *createEntry*
  `void` **createEntry**`( int` **arg0**`, java.lang.Object` **arg1**`, java.lang.Object` **arg2**`, int` **arg3** `)`
- *entrySet*
  `public Set` **entrySet**`( )`
- *get*
  `public Object` **get**`( java.lang.Object` **arg0** `)`
- *getEntry*
  `final HashMap.Entry` **getEntry**`( java.lang.Object` **arg0** `)`
- *hash*
  `static int` **hash**`( int` **arg0** `)`
- *indexFor*
  `static int` **indexFor**`( int` **arg0**`, int` **arg1** `)`
- *init*
  `void` **init**`( )`
- *isEmpty*
  `public boolean` **isEmpty**`( )`
- *keySet*
  `public Set` **keySet**`( )`
- *loadFactor*
  `float` **loadFactor**`( )`
- *newEntryIterator*
  `Iterator` **newEntryIterator**`( )`
- *newKeyIterator*
  `Iterator` **newKeyIterator**`( )`
- *newValueIterator*
  `Iterator` **newValueIterator**`( )`
- *put*
  `public Object` **put**`( java.lang.Object` **arg0**`, java.lang.Object` **arg1** `)`
- *putAll*
  `public void` **putAll**`( java.util.Map` **arg0** `)`
- *remove*
  `public Object` **remove**`( java.lang.Object` **arg0** `)`
- *removeEntryForKey*
  `final HashMap.Entry` **removeEntryForKey**`( java.lang.Object` **arg0** `)`

- *removeMapping*
  `final HashMap.Entry` **removeMapping**`( java.lang.Object` **arg0** `)`
- *resize*
  `void` **resize**`( int` **arg0** `)`
- *size*
  `public int` **size**`( )`
- *transfer*
  `void` **transfer**`( java.util.HashMap.Entry []` **arg0** `)`
- *values*
  `public Collection` **values**`( )`

METHODS INHERITED FROM CLASS `java.util.AbstractMap`

- *clear*
  `public void` **clear**`( )`
- *clone*
  `protected Object` **clone**`( )`
- *containsKey*
  `public boolean` **containsKey**`( java.lang.Object` **arg0** `)`
- *containsValue*
  `public boolean` **containsValue**`( java.lang.Object` **arg0** `)`
- *entrySet*
  `public abstract Set` **entrySet**`( )`
- *equals*
  `public boolean` **equals**`( java.lang.Object` **arg0** `)`
- *get*
  `public Object` **get**`( java.lang.Object` **arg0** `)`
- *hashCode*
  `public int` **hashCode**`( )`
- *isEmpty*
  `public boolean` **isEmpty**`( )`
- *keySet*
  `public Set` **keySet**`( )`
- *put*
  `public Object` **put**`( java.lang.Object` **arg0**`, java.lang.Object` **arg1** `)`
- *putAll*
  `public void` **putAll**`( java.util.Map` **arg0** `)`
- *remove*
  `public Object` **remove**`( java.lang.Object` **arg0** `)`
- *size*
  `public int` **size**`( )`
- *toString*
  `public String` **toString**`( )`
- *values*
  `public Collection` **values**`( )`

### C.2.6 CLASS **ComputationTreeCyclic**

## Declaration

public class ComputationTreeCyclic
**extends** alternation.runtime.ComputationTree

## Serializable Fields

- private Map nullEdges
  –

## Constructors

- *ComputationTreeCyclic*
  `public` **ComputationTreeCyclic**`( )`

## Methods

- *addNullEdge*
  `public boolean` **addNullEdge**`( alternation.runtime.ComputationNode` **parent**, `alternation.runtime.ComputationNode` **child** `)`
- *getParents*
  `public List` **getParents**`( alternation.runtime.ComputationNode` **child** `)`
- *removeNullEdges*
  `public void` **removeNullEdges**`( alternation.runtime.ComputationNode` **child** `)`

## Methods inherited from class `alternation.runtime.ComputationTree`

( in C.2.5, page 110)

- *addEdge*
  `public void` **addEdge**`( alternation.runtime.ComputationNode` **parent**, `alternation.runtime.ComputationNode` **child** `)`
  - **Usage**
    * Adds a edge between two Computation Nodes
  - **Parameters**
    * `parent` - fromNode
    * `child` - toNode
- *getChildren*
  `public List` **getChildren**`( alternation.runtime.ComputationNode` **node** `)`
  - **Parameters**
    * `node` - ComputationNode
  - **Returns** - Successors in the ComputationTree

- *getRoot*
  `public ComputationNode` **getRoot**`( )`
  - **Returns** - the root of the computation tree

---

- *getStateId*
  `public Byte` **getStateId**`( java.lang.String` **name** `)`
  - **Usage**
    - returns a states id for a given name
  - **Parameters**
    - `name` - state name
  - **Returns** - state id

---

- *getStateName*
  `public String` **getStateName**`( java.lang.Byte` **stateId** `)`
  - **Usage**
    - returns a states name for a given id
  - **Parameters**
    - `stateId` - state id
  - **Returns** - state name

---

- *put*
  `public Boolean` **put**`( alternation.runtime.ComputationNode` **node**`, java.lang.Boolean` **acceptance** `)`

---

- *setNameToStateId*
  `public void` **setNameToStateId**`( java.lang.String` **string**`, java.lang.Byte` **b** `)`
  - **Usage**
    - maps the state name with its id
  - **Parameters**
    - `string` - state name
    - `b` - state id

---

- *setRoot*
  `public void` **setRoot**`( alternation.runtime.ComputationNode` **root** `)`
  - **Parameters**
    - `root` - ComputationNode constucted by the input and the initial worktape

---

- *setStateIdToName*
  `public void` **setStateIdToName**`( java.lang.Byte` **b**`, java.lang.String` **string** `)`
  - **Usage**
    - maps the state id with its name
  - **Parameters**
    - `b` - state id
    - `string` - state name

## METHODS INHERITED FROM CLASS `java.util.HashMap`

- *addEntry*
  `void` **addEntry**`( int` **arg0**`, java.lang.Object` **arg1**`, java.lang.Object` **arg2**`, int` **arg3** `)`
- *capacity*
  `int` **capacity**`( )`
- *clear*
  `public void` **clear**`( )`
- *clone*
  `public Object` **clone**`( )`
- *containsKey*
  `public boolean` **containsKey**`( java.lang.Object` **arg0** `)`
- *containsValue*
  `public boolean` **containsValue**`( java.lang.Object` **arg0** `)`
- *createEntry*
  `void` **createEntry**`( int` **arg0**`, java.lang.Object` **arg1**`, java.lang.Object` **arg2**`, int` **arg3** `)`
- *entrySet*
  `public Set` **entrySet**`( )`
- *get*
  `public Object` **get**`( java.lang.Object` **arg0** `)`
- *getEntry*
  `final HashMap.Entry` **getEntry**`( java.lang.Object` **arg0** `)`
- *hash*
  `static int` **hash**`( int` **arg0** `)`
- *indexFor*
  `static int` **indexFor**`( int` **arg0**`, int` **arg1** `)`
- *init*
  `void` **init**`( )`
- *isEmpty*
  `public boolean` **isEmpty**`( )`
- *keySet*
  `public Set` **keySet**`( )`
- *loadFactor*
  `float` **loadFactor**`( )`
- *newEntryIterator*
  `Iterator` **newEntryIterator**`( )`
- *newKeyIterator*
  `Iterator` **newKeyIterator**`( )`
- *newValueIterator*
  `Iterator` **newValueIterator**`( )`
- *put*
  `public Object` **put**`( java.lang.Object` **arg0**`, java.lang.Object` **arg1** `)`
- *putAll*
  `public void` **putAll**`( java.util.Map` **arg0** `)`
- *remove*
  `public Object` **remove**`( java.lang.Object` **arg0** `)`
- *removeEntryForKey*
  `final HashMap.Entry` **removeEntryForKey**`( java.lang.Object` **arg0** `)`

- *removeMapping*
  `final HashMap.Entry` **removeMapping**`( java.lang.Object` **arg0** `)`
- *resize*
  `void` **resize**`( int` **arg0** `)`
- *size*
  `public int` **size**`( )`
- *transfer*
  `void` **transfer**`( java.util.HashMap.Entry []` **arg0** `)`
- *values*
  `public Collection` **values**`( )`

Methods inherited from class `java.util.AbstractMap`

- *clear*
  `public void` **clear**`( )`
- *clone*
  `protected Object` **clone**`( )`
- *containsKey*
  `public boolean` **containsKey**`( java.lang.Object` **arg0** `)`
- *containsValue*
  `public boolean` **containsValue**`( java.lang.Object` **arg0** `)`
- *entrySet*
  `public abstract Set` **entrySet**`( )`
- *equals*
  `public boolean` **equals**`( java.lang.Object` **arg0** `)`
- *get*
  `public Object` **get**`( java.lang.Object` **arg0** `)`
- *hashCode*
  `public int` **hashCode**`( )`
- *isEmpty*
  `public boolean` **isEmpty**`( )`
- *keySet*
  `public Set` **keySet**`( )`
- *put*
  `public Object` **put**`( java.lang.Object` **arg0**`, java.lang.Object` **arg1** `)`
- *putAll*
  `public void` **putAll**`( java.util.Map` **arg0** `)`
- *remove*
  `public Object` **remove**`( java.lang.Object` **arg0** `)`
- *size*
  `public int` **size**`( )`
- *toString*
  `public String` **toString**`( )`
- *values*
  `public Collection` **values**`( )`

### C.2.7 Class **ResultTuple**

Result of a state-computation on a worktape. So it saves if a configuration immediate accepts / rejects or its successors and if it is a universal or existential quantified configuration

#### Declaration

public class ResultTuple
**extends** java.lang.Object

#### Fields

- public static final ResultTuple ACCEPT
  - ResultTuple the means a computation accepts
- public static final ResultTuple REJECT
  - ResultTuple the means a computation rejects

#### Constructors

- *ResultTuple*
  `public` **ResultTuple**`( boolean` **accept**`, boolean` **reject**`, boolean` **forall** `)`
  - **Usage**
    - Constructor
  - **Parameters**
    - `accept` - true if the computation accepts
    - `reject` - true if the computation rejects
    - `forall` - if unversal quantified, false if existential quantified

#### Methods

- *add*
  `public void` **add**`( byte` **state**`, java.lang.Object` **worktape** `)`
  - **Usage**
    - adds a new Configuration to the Successors
  - **Parameters**
    - `state` - stateId
    - `worktape` - worktape

### C.2.8 Class **ResultTuple.AtmConfiguration**

simple container to save pairs of states and worktapes

#### Declaration

protected static class ResultTuple.AtmConfiguration
**extends** java.lang.Object

FIELDS

- public byte state
  –
- public Object worktape
  –

CONSTRUCTORS

- *ResultTuple.AtmConfiguration*
  `public` **ResultTuple.AtmConfiguration**`( byte` **state**`, java.lang.Object` **worktape** `)`

### C.2.9 CLASS **State**

abstract class representing a state of the alternating machine

DECLARATION

public abstract class State
**extends** java.lang.Object

FIELDS

- public Object atmInputtape
  –

CONSTRUCTORS

- *State*
  `public` **State**`( )`

METHODS

- *compute*
  `public abstract ResultTuple` **compute**`( java.lang.Object` **worktape** `)`
  - **Usage**
    - computes the Result from a worktape
  - **Parameters**
    - `worktape` - Work variables

  – **Returns** - Either accept / reject or a set of successors and a Quantifier (existential or universal).

---

- *setInput*
  `public void` **setInput**`( java.lang.Object` **input** `)`
  – **Parameters**
    * `input` - the actual input from the alternating machine

# Appendix D

# alternation.compiler

*Package Contents* *Page*

## D.1 Classes

### D.1.1 Class **atmCompile**

Gernerates Java code from an alternating algorithm

#### Declaration

public class atmCompile
**extends** java.lang.Object

#### Constructors

- *atmCompile*
  `public` **atmCompile**`( )`

#### Methods

- *compile*
  `public static void` **compile**`( java.lang.String` **inputPath**`, java.lang.String` **outputPath** `)`
  - **Usage**
    * Takes a .atm file and computes a java class
  - **Parameters**
    * `inputPath` - path to a .atm file
    * `outputPath` - path to save the output .java file
  - **Exceptions**
    * `java.io.IOException` -
    * `org.antlr.runtime.RecognitionException` -

- *main*
  `public static void` **main**`( java.lang.String []` **args** `)`

# Bibliography

[Aaronson et al., 2008] Aaronson, S., Kuperberg, G., and Granade, C. (2008). Complexity zoo. `http://www.complexityzoo.com`.

[Albert and Culik, 1987] Albert, J. and Culik, K. (1987). A simple universal cellular automaton and its one-way and totalistic version. *Complex Systems*, (1):1–16.

[Alvy Ray Smith, 1971] Alvy Ray Smith, I. (1971). Simple computation-universal cellular spaces. *J. ACM*, 18(3):339–353.

[Chandra et al., 1981] Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. (1981). Alternation. *J. ACM*, 28(1):114–133.

[Chandra and Tompa, 1990] Chandra, A. K. and Tompa, M. (1990). The complexity of short two-person games. *Discrete Appl. Math.*, 29(1):21–33.

[Cook, 1985] Cook, S. A. (1985). A taxonomy of problems with fast parallel algorithms. *Inf. Control*, 64(1-3):2–22.

[Fraenkel and Lichtenstein, 1981] Fraenkel, A. S. and Lichtenstein, D. (1981). Computing a perfect strategy for n*n chess requires time exponential in n. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 278–293, London, UK. Springer-Verlag.

[Greenlaw et al., 1995] Greenlaw, R., Hoover, H. J., and Ruzzo, W. L. (1995). *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, Inc., New York, NY, USA.

[Immerman, 1981] Immerman, N. (1981). Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences*, 22(3):384–406.

[Immerman, 1987] Immerman, N. (1987). Languages that capture complexity classes. *SIAM J. Comput.*, 16(4):760–778.

[Immerman, 1988] Immerman, N. (1988). Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938.

[Jones and Laaser, 1974] Jones, N. D. and Laaser, W. T. (1974). Complete problems for deterministic polynomial time. In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 40–46, New York, NY, USA. ACM.

[Kasai et al., 1979] Kasai, T., Adachi, A., and Iwata, S. (1979). Classes of pebble games and complete problems. *SIAM Journal on Compututing*, 8(4):574–586.

[Kasif, 1990] Kasif, S. (1990). On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artif. Intell.*, 45(3):275–286.

[Ladner, 1975] Ladner, R. E. (1975). The circuit value problem is log space complete for P. *SIGACT News*, 7(1):18–20.

[Lindgren and Nordahl, 1990] Lindgren, K. and Nordahl, M. G. (1990). Universal computation in simple one-dimensional cellular automata. *Complex Systems*, (4):299–318.

[Miyano et al., 1989] Miyano, S., Shiraishi, S., and Shoudai, T. (1989). A list of P - complete problems. *RIFIS Technical Report*, 17:1–69.

[Papadimitriou, 1994] Papadimitriou, C. M. (1994). *Computational complexity.* Addison-Wesley, Reading, Massachusetts.

[Parr, 2007] Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages.* The Pragmatic Bookshelf, Raleigh.

[Parr, 2008] Parr, T. (2008). Java 1.5 grammar for antlr v3. Website. `http://www.antlr.org/grammar/1152141644268//Java.g`.

[Plaisted, 1984] Plaisted, D. A. (1984). Complete problems in the first-order predicate calculus. In *Journal of Computer and System Sciences, 29:8–35.*

[Reisch, 1980] Reisch, S. (1980). Gobang ist PSPACE - vollständig. *Acta Inf.*, 13:59–66.

[Robson, 1983] Robson, J. M. (1983). The complexity of Go. In *IFIP Congress*, pages 413–417.

[Savitch, 1970] Savitch, W. J. (1970). Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192.

[Turing, 1936] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265.

[Venkateswaran, 1987] Venkateswaran, H. (1987). Properties that characterize LOGCFL. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 141–150, New York, NY, USA. ACM.

# Index