

Diploma Thesis

**Translating PSL into the Linear Time
 μ -Calculus**



Ludwig-Maximilians-Universität

München

Institut für Informatik

December 2006

Author: Marie-Fleur Revel

Supervisor: Dr. Martin Lange

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification.

Marie-Fleur Revel

Abstract

The Property Specification Language (PSL) as well as the linear time μ -calculus (μ TL) are temporal logics describing ω -regular languages. The syntax of μ TL contains fixed point operators whereas the definition of PSL combines regular expressions with formulas.

This thesis deals with the translation of PSL into the linear time μ -calculus in order to supply a more intuitive input language for applications based on μ TL. PSL formulas that do not contain regular expressions can directly be translated into μ TL in linear time and space complexity, depending on the syntactic length of the input formula. However, PSL formulas that include regular expressions cannot be translated directly and thus, the translation requires auxiliary means and techniques.

In this work, the translation from PSL to μ TL is defined with the aid of automata constructions. Therefore, translating a PSL formula into a μ TL formula requires exponential time and space in the number of intersection operators contained in the PSL formula.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Martin Lange. This thesis would not have been possible without his great support, remarkable patience and constructive comments.

Also, I thank my parents as well as Christian and Nina, who endured this process with me and offered their help whenever it was possible.

Contents

Declaration	ii
Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Temporal Logics	1
1.2 (Bounded) Model Checking	2
1.3 Outline of this Thesis	3
2 Preliminaries	4
2.1 Boolean Expressions	4
2.2 Automata	6
2.2.1 Nondeterministic Finite State Automata	6
2.2.2 Operations on NFAs	8
2.2.3 Nondeterministic Büchi Automata	13
2.3 Definition of PSL	14
2.3.1 Syntax	14
2.3.2 Semantics	16
2.3.3 Examples	18
2.4 Definition of μ TL	19
2.4.1 Syntax	19
2.4.2 Semantics	21
2.4.3 Examples	22
2.4.4 Model Checking Games over μ TL Formulas	22
2.5 Equation Systems	25
2.5.1 Definition	25
2.5.2 Model Checking Games over Equation Systems	26

2.6	Equivalence of μ TL Formulas and Equation Systems	28
2.6.1	Translating μ TL Formulas into Equation Systems	29
2.6.2	Translating Equation Systems into μ TL Formulas	32
3	Translating PSL to μTL	35
3.1	Preparations	35
3.1.1	Automata Constructions	35
3.1.2	Properties of μ TL and PSL Formulas	43
3.2	Translation and Proof	45
3.3	Complexity Analysis	47
4	Implementation	49
4.1	Framework	49
4.2	Data Types	49
4.2.1	PSL	49
4.2.2	Automata	50
4.2.3	Equation Systems	52
4.3	Functions	53
4.3.1	Translating Boolean and Regular Expressions to NFAs	53
4.3.2	Auxiliary Functions Reducing the Size of NFAs	55
4.3.3	Translating NFAs to Equation Systems	56
4.3.4	Translating PSL to Equation Systems	58
4.4	Empirical Results	62
4.4.1	N Bit Counter	62
4.4.2	Leap Years	65
4.4.3	Non-overlapping Repetition of Substrings in a String	67
5	Conclusion	70
5.1	Results Achieved	70
5.2	Outlook	70
A	Bibliography	72

1 Introduction

1.1 Temporal Logics

Temporal logic (TL) extends classical formal logic by allowing different logical values at different times or states of a system and was first introduced into computer science by Pnueli in 1977 [Pnu77]. Thus, TL represents a logical basis to formal treatment of state systems and is used to analyse properties of concurrent systems. Introductions to temporal logics in general can be found in [Eme90] and [Sti01].

The formal verification of systems - and therewith TL - is an important field of research in computer science because the correctness of hardware and software must be guaranteed and today's society totally trusts in it.

A famous example of verification being missed out is known as the Intel Pentium FDIV bug [Cor]. In 1994, an error in the Pentium floating point unit was discovered which caused economical damage and heavily affected the company's image.

In August 2006, erroneous software on a governmental Spanish internet server made over 400.000 internet sites disappear for hours, amongst others email and online banking accounts [FTD]. This very recent event shows the urgent need for research and development in the area of formal verification.

Temporal logic is usually separated into two different fields of study: *linear* and *branching* temporal logics. Linear temporal logic (LTL) works with representations of programs using linear structures like words; branching temporal logics utilizes branching structures, for example trees.

Just like LTL, the linear time μ -calculus (μ TL) describes properties of linear time structures but works with least and greatest fixed point operators. Whereas LTL describes star-free languages (first-order logic, as described in [Tho79] and in today's form in [Kam68] and [GPSS80]), μ TL describes the omega-regular languages (second-order logic). It emerged from the modal μ -calculus as introduced by Kozen in 1983 [Koz83] and was first presented in [Var88], [BKP86] and [BB89].

The μ -calculus provides a very powerful means of modelling properties of systems. Unfortunately, it also requires an unnegligible effort to understand the idea of fixed point logic characterizations. In contrast to that, the Property Specification Language [AO], which was introduced by the Accellera Organization Inc. and also describes omega-regular languages, is easier to handle. PSL extends LTL by including regular expressions into formulas. Its syntax and semantics will be explained in detail in section 2.3.

Kleene first deliniated regular expressions in order to describe properties of languages and sets. Regular expressions can be translated into automata and will play an important role in this work, amongst others because of constructions of automata over finite prefixes of infinite words, see chapter 3. An overview of the relations between logics and automata can be found in [Tho97] and [Var96].

1.2 (Bounded) Model Checking

Model checking defines the automated verification of systems, usually described with the aid of a transition system, with respect to some specification which is usually a temporal formula. It poses the question: If M is the description of a system and p a formula, does M satisfy p ? Clarke and Emerson as well as Queille and Sifakis established model checking in [CE82] and [JJ83].

Model checking faces a blow up of the size of transition systems, commonly known as the *state explosion problem*, that must be addressed to solve most real-world problems. One approach to reduce this complexity is *bounded* model checking [CBRZ01], which focusses on generating a counter-example of some maximal length k that disproves the satisfiability of the respective formula. Thus, systems of arbitrary size can be tested to the extend of an upper bound k . At the chair of theoretical computer science of the Ludwig Maximilians University in Munich, a verification tool named μ -Sabre [MuS] is being developed which can do bounded model checking for arbitrary regular properties.

The linear time μ -calculus has been chosen as the core language because its algorithmic handling is relatively easy. On the other hand, the application of μ TL as the input language is rather complicated because specifications are not quite intuitive. This is why we decided to translate a logic which is easier to handle and more intuitive - just like PSL - into the μ calculus.

The translation is based on the PSL core logic LTL_WR which is an extension of LTL. As

explained in [SBD05], LTL_{WR} and PSL are equi-expressive.

1.3 Outline of this Thesis

After defining the languages and automata used in this thesis in chapter 2, the translation of μ TL into PSL and its correctness will be explained in chapter 3. Just before the main proof, the reader will find additional definitions and lemmas that are needed to realize the main challenge of this work, namely the integration of regular expressions into formulas.

Chapter 4 gives an overview of the implementation and presents test cases in order to give the reader an impression of the advantages but also drawbacks of the implementation. Finally, this leads to the last chapter, the conclusions drawn from this thesis.

2 Preliminaries

2.1 Boolean Expressions

The set \mathbb{B} of boolean expressions is defined over a set \mathbb{P} of atomic propositions:

Definition 1 (*Boolean Expressions*) Let \mathbb{P} be a countable set of elements which are called *atomic propositions*. The set of **boolean expressions**, defined over \mathbb{P} , is described below.

- True and false are boolean expressions.
- Every atomic proposition p is a boolean expression.
- Given an atomic proposition p , its negation $\neg p$ is a boolean expression.
- Given boolean expressions φ and ψ , $(\varphi \vee \psi)$ and $(\varphi \wedge \psi)$ are boolean expressions.

Note. All languages and automata presented in this thesis are defined over the non-empty, finite alphabet $\Sigma = 2^{\mathbb{P}} \cup \{\top, \perp\}$ where \top means that any expression is interpreted as *true* and \perp declares everything as *false*.

Boolean expressions are defined in positive normal form throughout this work.

Definition 2 (*Boolean Satisfaction*) Let $a \in \Sigma^\omega$, p as well as q be atomic propositions and b be the boolean expression to be satisfied. The **boolean satisfaction** is defined as follows:

- $\top \models b \forall p$
- $\perp \not\models b \forall p$

If neither \top nor \perp applies,

- $a \models \text{true} \forall a$
- $a \not\models \text{false} \forall a$

- $a \models p \Leftrightarrow p \in b$
- $a \models \neg p \Leftrightarrow p \notin b$
- $a \models p \vee q \Leftrightarrow a \models p \text{ or } a \models q$
- $a \models p \wedge q \Leftrightarrow a \models p \text{ and } a \models q$

Definition 3 (Words) A **word** is a sequence of letters at which atomic propositions are, or are not fulfilled.

In this work, words will be represented as follows:

$$v = \begin{array}{l} v_1 \mid v_{1,0} \ v_{1,1} \ v_{1,2} \ \dots \\ v_2 \mid v_{2,0} \ v_{2,1} \ v_{2,2} \ \dots \\ \vdots \\ v_n \mid v_{n,0} \ v_{n,1} \ v_{n,2} \ \dots \end{array}$$

$v_{i,j} = 1$ means that the atomic proposition v_i is fulfilled at position j and $v_{i,j} = 0$ means it is not fulfilled.

$|v|$ denotes the length of a word v . We use v^i to refer to the $(i+1)^{st}$ letter of v , since the counting of letter starts at 0 and $v^{i..j}$ to describe the substring of v starting at v^i and finishing at v^j . $v^{i..}$ describes the suffix of v starting at i . If $v = v_1 v_2$, we say that v_1 is a prefix of v , denoted $v_1 \leq v$. Finally, the notation \bar{v} describes the word obtained by replacing every \top with a \perp and vice versa.

For infinite words, the notation $v_i \mid v_{i,0} \ v_{i,1} \ v_{i,2} \ \dots$ means that for $v_{i,j}$ such that $j > 2$, it is unknown if v_i is satisfied. In contrast to that, $v_i \mid v_{i,0} \ v_{i,1} \ (v_{i,2})^\omega$ describes that $v_{i,j} = v_{i,2}$ for all $j > 2$.

2.2 Automata

In order to translate regular expressions, two types of nondeterministic automata are needed, finite and infinite (Büchi automata).

2.2.1 Nondeterministic Finite State Automata

Definition 4 (*Nondeterministic Finite State Automaton*) Let $\neg\mathbb{P} := \{\neg p \mid p \in \mathbb{P}\}$. A nondeterministic finite state automaton (NFA) is a quintuple $(S, \Sigma, \delta, s_0, F)$, consisting of

- a finite set of states S
- a finite set of input symbols $\Sigma = 2^{\mathbb{P}} \cup \{\top, \perp\}$
- a transition function $\delta : S \rightarrow 2^{\mathbb{B}(\mathbb{P} \cup \neg\mathbb{P}) \times S}$
- an initial (or start) state $s_0 \in S$
- a finite set F of accepting (or final) states, $F \subseteq S$

Definition 5 (*Run*) Let $A = (S, \Sigma, \delta, s_0, F)$ be an NFA and w a finite word over Σ . A sequence of states q_0, q_1, \dots, q_n such that for all $i \in \{0, \dots, n\}$, there is a b such that $(b, q_{i+1}) \in \delta(q_i)$ and $w^i \models b$, is called **run**.

A run is **accepting**, if it reaches an accepting state.

Definition 6 (*NFA Acceptance*) Let A be an NFA such that $A = (S, \Sigma, \delta, s_0, F)$, and w be a finite word over the alphabet Σ . A accepts $w = w_0w_1w_2 \dots w_n$ iff there is a sequence of states $q_0q_1q_2 \dots q_n$ with $q_i \in S$ such that:

- $q_0 = s_0$
- $\forall i \exists b$ such that $(b, q_{i+1}) \in \delta(q_i)$ and $w_i \models b$
- $q_n \in F$

Definition 7 ($L(A)$) The **language** of words accepted by A is defined as

$$L(A) = \{w \mid A \text{ accepts } w\}$$

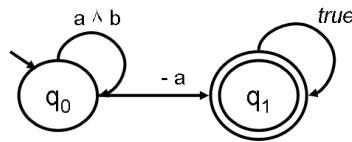


Figure 2.1: Example of an NFA

Examples

The automaton in figure 2.1 accepts all words over the alphabet Σ that start with arbitrary many positions at which a and b are fulfilled, directly followed by one position at which a is not fulfilled. It consists of the set of states $S = \{q_0, q_1\}$, where the incoming arrow determines the initial state q_0 and the final state q_1 is characterized by a double circle. All other arrows represent the transition function

$$\delta(q_0) = \{(q_0 \wedge a \wedge b), (\neg a \wedge q_1)\}$$

$$\delta(q_1) = \{\top \wedge q_1\}$$

The labels of transition functions, like $(a \wedge b)$, $\neg a$ or \top in this example, will often be referred to as *guards*.

1. Consider the word $v_1 = \begin{array}{l} a \mid 1 \ 1 \ 1 \ 1 \\ b \mid 1 \ 0 \ 0 \ 1 \end{array}$.

At the first position of v_1 , both a and b are satisfied, so that the NFA moves from q_0 to q_0 . Afterwards, a is fulfilled but not b - the automaton gets stuck. As it will never reach the final state, it does not accept v_1 .

2. The NFA does accept the word $v_2 = \begin{array}{l} a \mid 1 \ 1 \ 0 \ 1 \\ b \mid 1 \ 1 \ 1 \ 1 \end{array}$.

At the first two positions of v_2 , a and b are satisfied and therefore the automaton stays in q_0 . At the third position, a is not fulfilled and the automaton moves to the accepting state q_1 . From now on, it does not matter whether a and b are satisfied or not. As *true* declares everything as satisfied, the automaton will always stay in the accepting state q_1 .

2.2.2 Operations on NFAs

Nondeterministic finite state (NFA) as well as nondeterministic Büchi automata (NBA, see section 2.2.3) are closed under concatenation, union, intersection and Kleene star. These operations on automata will only be explained for NFAs in this chapter because NBAs are only needed as an intermediate step while translating PSL into μ TL and they will not be built inductively in this work.

Definition 8 (*Concatenation NFA*) If $A_1 = (S_1, \Sigma, \delta_1, s_{0_1}, F_1)$ and $A_2 = (S_2, \Sigma, \delta_2, s_{0_2}, F_2)$ are two NFAs such that $S_1 \cap S_2 = \emptyset$, the **concatenation NFA**

$$A_1; A_2 = (S_1 \cup S_2, \Sigma, \dot{\delta}, s_{0_1}, \dot{F})$$

has the same initial state as A_1 and connects the final states of A_1 to the initial states of A_2 . $A_1; A_2$ has the transition function

$$\bullet \dot{\delta}(q_i) = \begin{cases} \delta_1(q_i) & \text{if } q_i \in S_1 \text{ and } q_i \notin F_1 \\ \delta_2(q_i) & \text{if } q_i \in S_2 \\ \delta_1(q_i) \cup \delta_2(s_{0_2}) & \text{if } q_i \in F_1 \end{cases}$$

The set of final states is defined as

$$\bullet \dot{F} = \begin{cases} F_2 \cup F_1 & , \text{ if } s_{0_2} \in F_2 \\ F_2 & \text{ else.} \end{cases}$$

Lemma 1 $A_1; A_2$ accepts all words $v = v_1v_2$ such that A_1 accepts v_1 and A_2 accepts v_2 :

$$L(A_1; A_2) = L(A_1); L(A_2)$$

The proof can be found in [HU80].

Definition 9 (*Union NFA*) Let $A_1 = (S_1, \Sigma, \delta_1, s_{0_1}, F_1)$ and $A_2 = (S_2, \Sigma, \delta_2, s_{0_2}, F_2)$ be two NFAs, such that $S_1 \cap S_2 = \emptyset$ and $s_0 \notin S_1 \cup S_2$. The union NFA decides nondeterministically which automaton has to be used when starting to read input word. The formal definition of the **union NFA** is

$$A_1 \cup A_2 = (S_1 \cup S_2 \cup \{s_0\}, \Sigma, \delta, s_0, \dot{F})$$

where the new initial state s_0 points to the successors of the initial states of A_1 and A_2 :

- $\delta(s_0) = \delta_1(s_{0_1}) \cup \delta_2(s_{0_2})$

For all other states, the transition function δ works as follows:

- $\delta(q_i) = \begin{cases} \delta_1(q_i) & \text{if } q_i \in S_1 \\ \delta_2(q_i) & \text{otherwise} \end{cases}$

and the set of final states is defined as:

- $\dot{F} = \begin{cases} F_1 \cup F_2 \cup \{s_0\} & \text{, if } s_{0_1} \in F_1 \text{ or } s_{0_2} \in F_2 \\ F_1 \cup F_2 & \text{else.} \end{cases}$

Lemma 2 The union NFA $A_1 \cup A_2$ accepts all words that would either be accepted by A_1 or A_2 :

$$L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$$

The proof can be found in [HU80].

Definition 10 (*Intersection NFA*) Let $A_1 = (S_1, \Sigma, \delta_1, s_{0_1}, F_1)$ and $A_2 = (S_2, \Sigma, \delta_2, s_{0_2}, F_2)$ be two NFAs. The operation $(M_1 \times M_2)$ describes the cartesian product of the two sets M_1 and M_2 . The **intersection NFA** is defined as:

$$A_1 \cap A_2 = (\dot{S}, \Sigma, \dot{\delta}, s_0, \dot{F})$$

such that:

- $\dot{S} = S_1 \times S_2$ is the cartesian product of all states of A_1 and A_2 .
- $\dot{\delta} : \dot{S} \rightarrow 2^{(\mathbb{B}(\mathbb{P} \cup \neg \mathbb{P}) \times \dot{S})}$ links all combined states to those states that would be reached through δ_1 and δ_2 :

$$\delta(q_1, q_2) = \{(g_1 \wedge g_2, (q'_1, q'_2)) \mid (g_1, q'_1) \in \delta_1 \text{ and } (g_2, q'_2) \in \delta_2\}$$

In words, both guards g_1, g_2 must be fulfilled in order to move from (q_1, q_2) to (q'_1, q'_2) .

- $\dot{s}_0 = s_{0_1}, s_{0_2}$
- $\dot{F} = F_1 \times F_2$ is the cartesian product of states that are final in A_1 and A_2 .

Lemma 3 *The intersection NFA $A_1 \cap A_2$ accepts all words that would be accepted by both A_1 and A_2 :*

$$L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$$

The proof can be found in [HU80].

Definition 11 *(Kleene star NFA) For an NFA $A = (S, \Sigma, \delta, s_0, F)$, the **Kleene star NFA** is defined as:*

$$A^* = (S \cup \{\dot{s}_0\}, \Sigma, \dot{\delta}, \dot{s}_0, F \cup \{\dot{s}_0\})$$

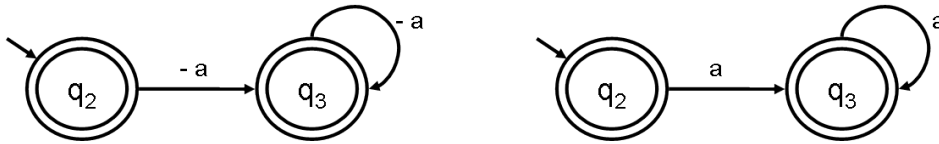
The Kleene Star NFA declares the initial state as an accepting state and adds all transitions of the initial state to the transitions of the final states:

$$\dot{\delta}(q_i) = \begin{cases} \delta(s_0) \cup \delta(q_i) & , \text{ if } q_i \in F \\ \delta(s_0) & , \text{ if } q_i = s_0 \\ \delta(q_i) & \text{ otherwise.} \end{cases}$$

Lemma 4 *The Kleene star NFA A^* accepts the empty word ϵ , all words that would be accepted by A and also all words accepted by A repeated arbitrarily often:*

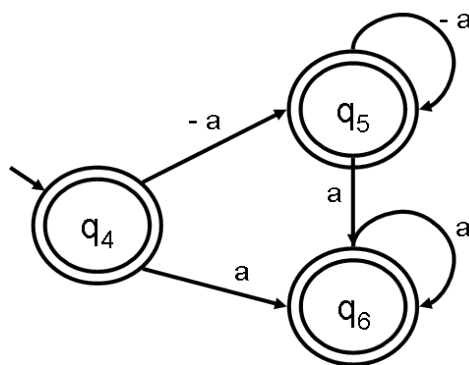
$$L(A^*) = (L(A))^*$$

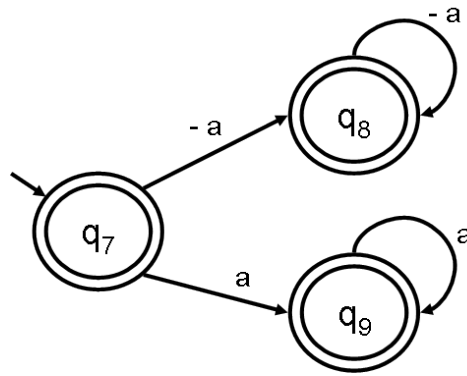
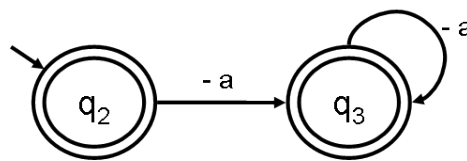
The proof can be found in [HU80].

Figure 2.2: NFAs A_1 and A_2 Figure 2.3: NFAs A_1^* and A_2^*

Examples

- Figure 2.2 shows the automata A_1 and A_2 , accepting words of length 1 whose first and only symbol satisfies $\neg a$ and a , respectively. Both move from the initial state q_0 to the final state q_1 when $\neg a$ or a are fulfilled.
- When applying the Kleene Star operation, we receive the automata A_1^* and A_2^* , see figure 2.3. As the initial state is also final, both automata accept the empty word. They also accept words that contain a sequence of arbitrary many symbols satisfying $\neg a$ or a respectively by moving to the second final state q_3 and staying there while the following symbols of the input word fulfil $\neg a$ or a . States that cannot reach the final state, such as the initial states of A_1 and A_2 , are removed from the new automata.
- The concatenation of A_1^* and A_2^* , namely A_3 , is shown in figure 2.4. As both automata accept the empty word, the concatenation NFA also does. Furthermore, words consist-

Figure 2.4: NFA $A_3 = A_1^*; A_2^*$

Figure 2.5: NFA $A_4 = A_1^* \cup A_2^*$ Figure 2.6: NFA $A_5 = A_4 \cap A_1^*$

ing of a sequence of arbitrary many symbols satisfying a are accepted, which can be interpreted as A_1^* accepting the empty word followed by A_2^* accepting the sequence of symbols fulfilling a . Similarly, words that consist of a sequence of symbols fulfilling $\neg a$ are accepted. Finally, if a word starts with a sequence of symbols satisfying $\neg a$ and ends with a sequence of symbols satisfying a , the automaton accepts it by moving from q_3 to q_4 while reading $\neg a$ and then reaching q_5 when the first a occurs.

4. Figure 2.5 shows the union NFA $A_4 = A_1^* \cup A_2^*$. The automaton accepts again the empty word because A_1^* as well as A_2^* do. Apart from that, words that contain a sequence of symbols fulfilling $\neg a$ or of a are accepted by moving to and staying in the final states q_8 or q_9 , respectively. This describes exactly the words that would be accepted by either A_1^* or A_2^* .
5. Figure 2.6 shows the intersection NFA ($NFA_5 = A_4 \cap A_1^*$). This automaton accepts the same words as A_1^* , because A_4 accepts all words that would be accepted by A_1^* but A_1^* does not accept words that contain a sequence of a .

2.2.3 Nondeterministic Büchi Automata

In contrast to NFAs which describe regular languages of finite words, Büchi automata describe ω -regular languages. They were introduced by Büchi in [Büc62].

In the PSL to μ TL translation, Büchi automata are needed as an auxiliary means to interpret the combination of PSL regular expressions and PSL formulas, see definition 15.

Definition 12 (*Nondeterministic Büchi Automaton*) A **nondeterministic Büchi automaton (NBA)** is a quintuple $(S, \Sigma, \delta, s_0, F)$, consisting of

- a finite set of states S
- a finite set of input symbols $\Sigma = 2^{\mathbb{P}} \cup \{\top, \perp\}$
- a transition function $\delta : S \rightarrow 2^{(\mathbb{B}(\mathbb{P} \cup \neg\mathbb{P}) \times S)}$
- an initial (or start) state $s_0 \in S$
- a finite set F of accepting (or final) states, $F \subseteq S$

Definition 13 (*NBA Acceptance*) Let B be an NBA such that $B = (S, \Sigma, \delta, s_0, F)$, and w be an infinite word over the alphabet Σ . B accepts $w = w_0w_1w_2 \dots$ iff there is a sequence of states $q_0q_1q_2 \dots$ with $q_i \in S$ such that:

- $q_0 = s_0$
- $\forall i \exists b$ such that $(b, q_{i+1}) \in \delta(q_i)$ and $w_i \models b$
- $q_i \in F$ for infinitely many i

Examples

Consider the following Büchi automaton in figure 2.7, given by

- the set of states $S = \{q_0, q_1\}$
- the transition function

$$\delta(q_0) = \{(q_0 \wedge \top), (q_1 \wedge \neg a)\}$$

$$\delta(q_1) = \{(q_0 \wedge a)\}$$

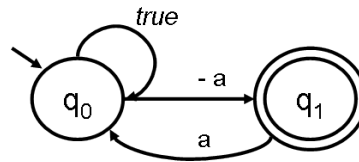


Figure 2.7: Example of an NBA

- the initial state q_0
- and the final state q_1 .

This automaton accepts all words that contain infinitely many sections of the form uv , such that $a \in u$ and $a \notin v$.

1. Let $v_1 = a | 1 1 0 1 1 (1)^\omega$

The automaton will first stay in q_0 and then move to q_1 at the third position of v_1 when reading $-a$. After that, it will move back to q_0 and stay there, because a is always fulfilled. Thus, the automaton only reaches q_1 once and not infinitely often. It does not accept v_1 .

2. The word $v_2 = a | (1 1 0)^\omega$

is accepted by the automaton, because at every sequence $(-a, a)$, it moves to q_1 and back. In this way, the final state q_1 will be reached infinitely often.

3. $v_3 = a | (0 0 1)^\omega$

is also accepted by the Büchi automaton in figure 2.7. It will first stay in q_0 and then move to q_1 at the second position of v_3 , which is a nondeterministic guess. At the third position of v_3 , the automaton moves back to q_0 . This will happen repeatedly and make the automaton reach the final state infinitely often.

2.3 Definition of PSL

Below the reader will find the definition of the PSL core logic LTL_WR , an extension of LTL , interpreted over finite and infinite words. For convenience, LTL_WR will often be referred to as PSL in this work.

2.3.1 Syntax

PSL is defined over boolean expressions, regular expressions and formulas.

Definition 14 (*Regular Expressions*) **Regular expressions (RE)** are defined over the set of boolean expressions.

- Every boolean expression $b \in \mathbb{B}$ is an RE.
- The empty word ϵ is an RE.
- If r, r_1 and r_2 are REs, then the following are REs:
 - $r_1 ; r_2$
(Concatenation over regular expressions)
 - $r_1 \cup r_2$
(Union over regular expressions)
 - $r_1 \cap r_2$
(Intersection of regular expressions)
 - r^*
(Kleene Star over a regular expression)

Note. In different contexts the definition of regular expressions will often include \emptyset , the empty set. However, it is not part of this work's definition because it can be modelled with the help of the intersection operator.

Definition 15 (*PSL Formulas*)

- If b is a boolean expression then $b!$ is an PSL formula.
- Every regular expression r is a formula.
- If φ and ψ are PSL formulas, and r is a RE, then the following are PSL formulas:
 - $\neg\varphi$
(Negation of a formula)
 - $\varphi \vee \psi$
(Disjunction over formulas)
 - $X! \varphi$
(Next operator over a formula)
 - $\varphi U \psi$
(Until over formulas)

– $r \diamondrightarrow \varphi$

(Next operator over a formula and a regular expression)

Definition 16 (Subformulas) *The set of **subformulas** $Sub(\varphi)$ of a PSL formula is defined as follows:*

- for all boolean expressions b, b_1, b_2 :

$$Sub(b!) := \begin{cases} \{true\} & , \text{ if } b = true \\ \{false\} & , \text{ if } b = false \\ \{p\} & , \text{ if } b = p \in \mathbb{P} \\ \{b_1 \vee b_2\} \cup Sub(b_1) \cup Sub(b_2) & , \text{ if } b = b_1 \vee b_2 \\ \{b_1 \wedge b_2\} \cup Sub(b_1) \cup Sub(b_2) & , \text{ if } b = b_1 \wedge b_2 \end{cases}$$

- for regular expressions r, r_1 and r_2 , we define:

$$Sub(r) := \begin{cases} \{\epsilon\} & , \text{ if } r = \epsilon \\ \{r_1; r_2\} \cup Sub(r_1) \cup Sub(r_2) & , \text{ if } r = r_1; r_2 \\ \{r_1 \cup r_2\} \cup Sub(r_1) \cup Sub(r_2) & , \text{ if } r = r_1 \cup r_2 \\ \{r_1 \cap r_2\} \cup Sub(r_1) \cup Sub(r_2) & , \text{ if } r = r_1 \cap r_2 \\ \{r_1^*\} \cup Sub(r_1) & , \text{ if } r = r_1^* \end{cases}$$

- $Sub(\neg\varphi) := \{\neg\varphi\} \cup Sub(\varphi)$
- $Sub(\varphi \vee \psi) := \{\varphi \vee \psi\} \cup Sub(\varphi) \cup Sub(\psi)$
- $Sub(X!\varphi) := \{X!\varphi\} \cup Sub(\varphi)$
- $Sub(\varphi U \psi) := \{\varphi U \psi\} \cup Sub(\varphi) \cup Sub(\psi)$
- $Sub(r \diamondrightarrow \varphi) = \{r \diamondrightarrow \varphi\} \cup Sub(r) \cup Sub(\varphi)$

2.3.2 Semantics

In PSL, there are three different satisfaction relations:

- \models : boolean satisfaction (see section 2.1)
- \models_{PSL} : RE tight satisfaction

- \vDash_{PSL} : formula satisfaction

Boolean satisfaction defines in which case a symbol of a respective word fulfils a boolean expression. RE tight satisfaction and formula satisfaction provide the corresponding definitions for finite words and infinite words, fulfilling regular expressions and formulas, respectively.

Definition 17 (*RE Tight Satisfaction*) *Let v denote a finite word over Σ , b a boolean expression and r, r_1, r_2 regular expressions. The **RE tight satisfaction** is defined as follows:*

- $v \models_{PSL} \epsilon$ if v is the empty word
- $v \models_{PSL} b \Leftrightarrow |v| = 1$ and $v^0 \models b$
- $v \models_{PSL} r_1 ; r_2 \Leftrightarrow \exists v_1, v_2$ such that $v = v_1 v_2$ and $v_1 \models_{PSL} r_1$ and $v_2 \models_{PSL} r_2$
- $v \models_{PSL} r_1 \cup r_2 \Leftrightarrow v \models_{PSL} r_1$ or $v \models_{PSL} r_2$
- $v \models_{PSL} r_1 \cap r_2 \Leftrightarrow v \models_{PSL} r_1$ and $v \models_{PSL} r_2$
- $v \models_{PSL} r^* \Leftrightarrow \exists k \in \mathbb{N}$ s.t. $v \models_{PSL} r^k$ where $r^0 = \epsilon$, $r^{k+1} = r; r^k$

Later in this work, we will refer to the language defined by a regular expression as $L(r)$, formally: $L(r) := \{w \mid w \models_{PSL} r\}$.

Definition 18 (*PSL Formula Satisfaction*) *For an infinite word v over Σ , a boolean expression b , a regular expression r and PSL formulas φ and ψ , the **PSL formula satisfaction** is explained below.*

- $v \vDash_{PSL} b! \Leftrightarrow v^0 \models b$, see definition 2.
- $v \vDash_{PSL} \neg\varphi \Leftrightarrow \bar{v} \not\vDash_{PSL} \varphi$
- $v \vDash_{PSL} \varphi \vee \psi \Leftrightarrow v \vDash_{PSL} \varphi$ or $v \vDash_{PSL} \psi$
- $v \vDash_{PSL} X!\varphi \Leftrightarrow v^{1..} \vDash_{PSL} \varphi$
- $v \vDash_{PSL} \varphi U \psi \Leftrightarrow \exists k \in \mathbb{N} : v^{k..} \vDash_{PSL} \psi$ and $\forall j < k$ such that $v^{j..} \vDash_{PSL} \varphi$

- $v \models_{PSL} r \diamond \rightarrow \varphi \Leftrightarrow \exists j \in \mathbb{N}$ such that $v^{0..j} \models_{PSL} r$ and $v^{j..} \models_{PSL} \varphi$
- $v \models_{PSL} r \Leftrightarrow \forall$ finite $u \leq v$, $u \top^\omega \models_{PSL} r \diamond \rightarrow true$

Note. The formula $\varphi \wedge \psi$, meaning that both φ and ψ must hold, will be used as an abbreviation for $\neg((\neg\varphi) \vee (\neg\psi))$.

2.3.3 Examples

Similar to words that are defined by atomic propositions, we define words over the formulas $\varphi_1 \dots \varphi_n$ as follows, meaning φ_i is fulfilled at position j , if $\varphi_{i,j} = 1$:

$$v = \begin{array}{l} \varphi_1 \mid \varphi_{1,1} \varphi_{1,2} \varphi_{1,3} \dots \\ \varphi_2 \mid \varphi_{2,1} \varphi_{2,2} \varphi_{2,3} \dots \\ \vdots \\ \varphi_n \mid \varphi_{n,1} \varphi_{n,2} \varphi_{n,3} \dots \end{array}$$

1. Let p_1 and p_2 be atomic propositions. The boolean expression $b = (p_1 \vee p_2)$ means that p_1 or p_2 must be satisfied. If we define two words

$$v_1 = \begin{array}{l} p_1 \mid 0 \\ p_2 \mid 1 \end{array} \text{ and } v_2 = \begin{array}{l} p_1 \mid 0 \\ p_2 \mid 0 \end{array} \text{ then}$$

- $v_1 \models b$ because p_2 is fulfilled, but
- $v_2 \not\models b$ because neither b_1 nor b_2 are satisfied.

2. If b_1, b_2 and b_3 are boolean expressions, the regular expression $r = (b_1 \wedge b_2); b_3^*$ defines that b_1 and b_2 must hold at the first position of a word and afterwards, arbitrary many b_3 must follow. This is why, for the words

$$v_3 = \begin{array}{l} b_1 \mid 1 \ 1 \ 0 \\ b_2 \mid 1 \ 0 \ 0 \\ b_3 \mid 1 \ 1 \ 1 \end{array} \quad v_4 = \begin{array}{l} b_1 \mid 1 \\ b_2 \mid 1 \\ b_3 \mid 0 \end{array} \quad \text{and} \quad v_5 = \begin{array}{l} b_1 \mid 1 \ 1 \ 1 \\ b_2 \mid 0 \ 1 \ 1 \\ b_3 \mid 1 \ 1 \ 1 \end{array}$$

the following holds:

- $v_3 \models_{PSL} r$ and
- $v_4 \models_{PSL} r$ because the second part of the formula, namely b_3^* is also satisfied by the empty word;
- $v_5 \not\models_{PSL} r$ because b_2 is not satisfied at the first position of v_5 .

3. Let φ and ψ be PSL formulas. The formula $\tau = \varphi \wedge (X!(\psi U \varphi))$ describes that after φ holds at the beginning of a word, ψ must hold until φ holds again. We define the words

$$v_7 = \begin{array}{l} \varphi \mid 1\ 1\ 1\ (1)^\omega \\ \psi \mid 0\ 0\ 0\ (0)^\omega \end{array} \quad v_8 = \begin{array}{l} \varphi \mid 1\ 0\ 0\ (1)^\omega \\ \psi \mid 0\ 1\ 1\ (0)^\omega \end{array} \quad \text{and} \quad v_9 = \begin{array}{l} \varphi \mid 1\ 0\ 0\ (0)^\omega \\ \psi \mid 0\ 1\ 1\ (1)^\omega \end{array} .$$

It is now easy to see that:

- $v_7 \not\models \tau$ because φ is never read
- $v_8 \models \tau$ because the word starts with φ and after ψ occurs, φ holds again.
- As the definition of $X!(\varphi U \psi)$ specifies that φ must be read at any finite position of v_9 bigger than 0, $v_9 \not\models \tau$.

2.4 Definition of μ TL

The linear time μ -calculus extends LTL by defining fixed point operators and is explained, for example, in [Kai97].

2.4.1 Syntax

Definition 19 (*μ TL Formulas*) Let $\Sigma = 2^{\mathbb{P}} \cup \{\top, \perp\}$ be a finite non-empty alphabet and $V = \{X, Y, Z, \dots\}$ be a set of variables. The following are μ TL formulas:

- $p \in \mathbb{P}$
(atomic propositions)
- $\neg p$, where $p \in \mathbb{P}$
(negated atomic propositions)
- $X \in V$
(Variables)

If φ and ψ are μ TL formulas, then the following are also μ TL formulas:

- $\varphi \vee \psi$
(Disjunction over formulas)
- $\varphi \wedge \psi$
(Conjunction over formulas)
- $\bigcirc \varphi$
(Next operator over a formula)

- $\mu X.\varphi$
(Least fixed point operator)
- $\nu X.\varphi$
(Greatest fixed point operator)

Later in this work, the μ - or ν - binder will be referred to as σ .

Definition 20 (Subformulas) *The set of **subformulas** $Sub(\varphi)$ of a μTL formula φ is defined as:*

- $Sub(p) := \{p\}, \forall p \in \mathbb{P}$
- $Sub(\neg p) := \{\neg p\}, \forall p \in \mathbb{P}$
- $Sub(X) := \{X\}, \forall X \in V$
- $Sub(\varphi \vee \psi) := \{\varphi \vee \psi\} \cup Sub(\varphi) \cup Sub(\psi)$
- $Sub(\varphi \wedge \psi) := \{\varphi \wedge \psi\} \cup Sub(\varphi) \cup Sub(\psi)$
- $Sub(\bigcirc \varphi) := \{\bigcirc \varphi\} \cup Sub(\varphi)$
- $Sub(\mu X.\varphi) := \{\mu X.\varphi\} \cup Sub(\varphi)$
- $Sub(\nu X.\varphi) := \{\nu X.\varphi\} \cup Sub(\varphi)$

Definition 21 (Bound variables) *The occurrence of a variable X in a μTL formula φ is **bound**, iff there is a $\sigma X.\psi \in Sub(\varphi)$ such that $X \in Sub(\psi)$. Otherwise, it is **free**. A formula is **closed** iff it has no free variables.*

Note. We assume that variables in a μTL formula can only be bound once, which makes it possible to assure the uniqueness of formulas of the type $\sigma X.\varphi$.

Definition 22 ($<_{\varphi}$) *For two variables $X, Y \in Sub(\varphi)$ we write $X <_{\varphi} Y$ iff Y is free in some $\sigma X.\psi \in Sub(\varphi)$.*

Definition 23 (Alternation-free formulas) *A μTL formula ψ is called **alternation-free** if it contains no subformula of the type $\sigma X.\varphi(Y)$ such that Y is free in φ but bound to some $\tilde{\sigma}$ such that $\tilde{\sigma} \neq \sigma$.*

Example: Consider the following non-alternation-free μ TL formula φ :

$$\varphi = \mu X. \nu Y. (a \wedge \bigcirc Y) \vee (b \wedge \bigcirc X)$$

X is free in ν but bound in μ and thus, the formula is alternating. In contrast to that, the following formula ψ is alternation-free, because X is not free in ν anymore:

$$\psi = \mu X. (\nu Y. a \wedge \bigcirc Y) \vee (b \wedge \bigcirc X).$$

2.4.2 Semantics

Definition 24 (*μ TL Formula Satisfaction*) *The semantics of a μ TL formula is defined over infinite words $w \in \Sigma^\omega$. Formulas with free variables are interpreted with respect to an environment $\rho : V \rightarrow 2^{\mathbb{N}}$ which maps all free variables to positions $S \subseteq \mathbb{N}$. The notation $\rho[Y := S]$ means that only the mapping of Y is changed to S in ρ . **Formula satisfaction** for words over $\tilde{\Sigma} := \Sigma \setminus \{\top, \perp\}$ is defined as follows:*

- $\llbracket p \rrbracket_\rho^w := \{i \in \mathbb{N} \mid p \in w^i\}$
- $\llbracket \neg p \rrbracket_\rho^w := \{i \in \mathbb{N} \mid p \notin w^i\}$
- $\llbracket X \rrbracket_\rho^w := \rho(X)$
- $\llbracket \varphi \vee \psi \rrbracket_\rho^w := \llbracket \varphi \rrbracket_\rho^w \cup \llbracket \psi \rrbracket_\rho^w$
- $\llbracket \varphi \wedge \psi \rrbracket_\rho^w := \llbracket \varphi \rrbracket_\rho^w \cap \llbracket \psi \rrbracket_\rho^w$
- $\llbracket \bigcirc \varphi \rrbracket_\rho^w := \{i \in \mathbb{N} \mid i + 1 \in \llbracket \varphi \rrbracket_\rho^w\}$
- $\llbracket \mu X. \varphi \rrbracket_\rho^w := \bigcap \{S \subseteq \mathbb{N} \mid \llbracket \varphi \rrbracket_{\rho[X:=S]}^w \subseteq S\}$
- $\llbracket \nu X. \varphi \rrbracket_\rho^w := \bigcup \{S \subseteq \mathbb{N} \mid S \subseteq \llbracket \varphi \rrbracket_{\rho[X:=S]}^w\}$

Finally, words over $\Sigma = 2^{\mathbb{P}} \cup \{\top, \perp\}$ are interpreted as

- $\llbracket \varphi \rrbracket_\rho^w = (\llbracket \varphi \rrbracket_\rho^w \cup \{i \mid w_i = \top\}) \setminus \{i \mid w_i = \perp\}$

Later in this work, the notation $v \vDash_{\mu TL} \varphi$ will be used to express that a word v satisfies the closed μ TL formula φ for arbitrary ρ :

$$v \vDash_{\mu TL} \varphi \Leftrightarrow 0 \in \llbracket \varphi \rrbracket_\rho^v$$

2.4.3 Examples

1. If p_1 , p_2 and p_3 are atomic propositions, the μ TL formula $\varphi = (p_1 \vee p_2) \wedge (\bigcirc p_3)$ describes that at the beginning of a word, p_1 or p_2 must hold and at the next position, p_3 must hold.

It is easy to see that, for the words

$$\begin{array}{lll} p_1 \mid 1\ 0\ 1\ 0\ \dots & p_1 \mid 0\ 0\ 0\ 0\ \dots & p_1 \mid 0\ 1\ 0\ 1\ \dots \\ v_1 = p_2 \mid 0\ 0\ 0\ 0\ \dots & v_2 = p_2 \mid 1\ 0\ 0\ 0\ \dots & v_3 = p_2 \mid 0\ 0\ 0\ 0\ \dots \\ p_3 \mid 0\ 1\ 0\ 1\ \dots & p_3 \mid 0\ 1\ 1\ 1\ \dots & p_3 \mid 1\ 0\ 1\ 0\ \dots \end{array}$$

- $v_1 \models_{\mu TL} \varphi$ and
 - $v_2 \models_{\mu TL} \varphi$ because in both v_1 and v_2 , p_1 or p_2 are satisfied at the first position, followed by p_3 at the second position of the word.
 - $v_3 \not\models_{\mu TL} \varphi$ because at the first position of v_3 , neither p_1 nor p_2 are satisfied.
2. The μ TL formula $\psi = \nu X.((p_1 \vee p_2) \wedge \bigcirc X)$ determines that at all positions of a word, either p_1 or p_2 must be satisfied. The words

$$v_4 = \begin{array}{l} p_1 \mid (1\ 1\ 1\ 1)^\omega \\ p_2 \mid (0\ 0\ 0\ 0)^\omega \end{array} \quad \text{and} \quad v_5 = \begin{array}{l} p_1 \mid (0\ 1\ 0\ 1)^\omega \\ p_2 \mid (1\ 0\ 1\ 0)^\omega \end{array} \quad \text{both satisfy this condition:}$$

- $v_4 \models_{\mu TL} \psi$ and
- $v_5 \models_{\mu TL} \psi$.

2.4.4 Model Checking Games over μ TL Formulas

As logical games will play an important role in some of the proofs in this thesis, this section will give the reader an introduction to these games and their appropriate strategies. They were first introduced for the modal μ -calculus in [Sti95] and are explained for the linear time μ -calculus in [Dax06], for example.

Definition 25 (*Play*) Let p_1 and p_2 be players and R be a set of rules. A **play** P is a finite or infinite sequence of configurations $P = C_0 C_1 C_2 \dots$ such that $\forall i \in \mathbb{N} : (C_i, C_{i+1})$ is an instance of some rule $r \in R$. If $D = C_i$ is the current configuration in a play, then player p_j can perform his choice which will lead to the next configuration, namely C_{i+1} which is of the form D_i . This will be represented in this thesis as follows:

$$r \frac{D}{D_1 \dots D_n} p_j \text{ where } j \in \{1, \dots, m\}$$

Definition 26 (*Memoryless strategy*) A **strategy** for a player p defines all his possible choices at every configuration C_i of a play. A winning, memoryless strategy is a strategy that leads to

victory in a game, regardless of the other player's actions. We abbreviate "p has a winning, memoryless strategy for a game" by saying "p wins a game".

Note. As proved in [EJ91], memoryless strategies are sufficient for issues discussed in this work.

Definition 27 (*fp*) Let φ be a μ TL formula and Var be a set of variables. We define the function $fp_\varphi : Sub(\varphi) \cap Var \rightarrow Sub(\varphi)$ which maps each X to its unique binder: $fp_\varphi(X) = \sigma X.\psi$

Definition 28 (*Model checking games over μ TL formulas*) Let $w \in \Sigma^\omega$ be a word and φ be a μ TL formula. A **model checking game** $G(v, \varphi)$ between the two players \exists , called Elizabeth, and \forall , called Albert, consists of

- a set of configurations $C = \{(w^{i..}, \psi) \mid i \in \mathbb{N} \text{ and } \psi \in Sub(\varphi)\}$
- an initial configuration $C_0 = (w, \varphi)$
- the following set of rules:

$$\begin{array}{ll}
 (\vee) \frac{w^{i..}, \varphi_0 \vee \varphi_1}{w^{i..}, \varphi_k} & (\exists), k \in \{0, 1\} \quad (\wedge) \frac{w^{i..}, \varphi_0 \wedge \varphi_1}{w^{i..}, \varphi_k} \quad (\forall), k \in \{0, 1\} \\
 (\mu) \frac{w^{i..}, \mu X.\varphi}{w^{i..}, X} & (\nu) \frac{w^{i..}, \nu X.\varphi}{w^{i..}, X} \\
 (\circ) \frac{w^{i..}, \circ\varphi}{w^{i+1..}, \varphi} & (X) \frac{w^{i..}, X}{w^{i..}, \psi}, \text{ if } fp_\varphi(X) = \sigma X.\psi
 \end{array}$$

Let φ_1 and φ_2 be subformulas of φ . If a configuration of the type $(w^{i..}, (\varphi_1 \vee \varphi_2))$ occurs, player \exists may choose between the disjuncts φ_1 and φ_2 . In situations of the sort $(w^{i..}, (\varphi_1 \wedge \varphi_2))$, it is \forall 's turn to choose. In all other cases, the next configuration is reached through a deterministic step according to the other rules.

- *Winning conditions:*

Player \exists wins a play if

- the play ends with $C_n = (w^{i..}, p)$ where $p \in \mathbb{P}$ and $w^i \vDash_{\mu TL} p$,
- if the greatest variable with respect to $<_\varphi$ occurring infinitely often is of type ν .

Player \forall wins a play if

- the play ends with $C_n = (w^{i..}, p)$ where $p \in \mathbb{P}$ and $w^i \not\models_{\mu TL} p$,
- if the greatest variable with respect to $<_{\varphi}$ occurring infinitely often is of type μ .

Theorem 1 (Model checking games and satisfiability of μTL formulas) *Let φ be a closed μTL formula and w a word in Σ^{ω} . The following holds:*

- Player \exists wins the model checking game $G(w, \varphi) \Leftrightarrow w \models_{\mu TL} \varphi$
- Player \forall wins the model checking game $G(w, \varphi) \Leftrightarrow w \not\models_{\mu TL} \varphi$

The proof can be found in [Sti01] .

Examples

Let a, b and c be in \mathbb{P} . Imagine \forall , Albert, and \exists , Elizabeth, playing a model checking game

$$a \mid (1\ 0)^{\omega}$$

over the word $w = b \mid (0\ 1)^{\omega}$ with respect to different formulas in the examples below.

$$c \mid (0\ 0)^{\omega}$$

1. $w \stackrel{?}{\models}_{\mu TL} a \wedge \bigcirc(b \vee c)$

First, it's \forall 's turn, he has to play $(a \wedge \bigcirc(b \vee c))$. As a is fulfilled and the satisfaction of the second part of the formula, namely $\bigcirc(b \vee c)$, is yet unknown, \forall chooses $\bigcirc(b \vee c)$. By choosing a , he would have lost the play right here; $\bigcirc(b \vee c)$ still leaves him a chance of proving that the formula is *not* satisfied.

For $\bigcirc(b \vee c)$, we need a deterministic step and \exists 's rule: First, we move to the next position of the word using the rule (\bigcirc) . Then, \exists plays (\vee) : She can choose between b or c ; as she wants to show that the formula is satisfied, she chooses b and therefore wins the play.

The fact that \exists would win the game irrespective of \forall 's actions proves that $w \models_{\mu TL} a \wedge \bigcirc(b \vee c)$.

2. $w \stackrel{?}{\models}_{\mu TL} \nu X.a \wedge \bigcirc(b \wedge \bigcirc X)$

This play starts with the steps (ν) and (X) , which leads to $a \wedge \bigcirc(b \wedge \bigcirc X)$. Now, \forall chooses $\bigcirc(b \wedge \bigcirc X)$ because a is fulfilled. After using the (\bigcirc) rule, it is again \forall 's turn and he will choose $\bigcirc X$, as b is satisfied at the current position of the word. The situation after \forall 's last move is the same as in the beginning: we are in a loop.

the greatest variable with respect to $<_{\varphi}$ occurring infinitely often is of type ν , \exists wins regardless of \forall 's choices which shows us that $w \vDash_{\mu\text{TL}} \nu X.a \wedge \bigcirc(b \wedge \bigcirc X)$.

3. $w \vDash_{\mu\text{TL}}^? \mu X.c \vee ((a \vee b) \wedge \bigcirc X)$

This example leads to an infinite play: First, the steps (μ) and (X) lead to $c \vee ((a \vee b) \wedge \bigcirc X)$ and afterwards, \exists plays $((a \vee b) \wedge \bigcirc X)$ as she will never read c . As $(a \vee b)$ will be true at all positions of w , \forall chooses $\bigcirc X$ which leads again to $c \vee ((a \vee b) \wedge \bigcirc X)$; we are in a loop. As the greatest variable with respect to $<_{\varphi}$ occurring infinitely often is of type μ , \forall wins.

This proves that $w \not\vDash_{\mu\text{TL}} \mu X.c \vee ((a \vee b) \wedge \bigcirc X)$.

2.5 Equation Systems

Later in this thesis, the translation from PSL to μTL is realized via *equation systems*, which are equivalent to μTL expressions (see section 2.6). The reason is that equation systems provide a compact representation of μTL -formulas. Considering the fact that subformulas of a μTL formula φ might occur repeatedly, it is easy to see that there are representations of φ that can be exponentially shorter than the syntactic length of φ , such as equation systems which number all subformulas and can realize a sharing over those occurring more than once.

2.5.1 Definition

Definition 29 (*Equation Systems*) Let \mathbb{P} be the set of atomic propositions and V be a finite set of variables. An **equation system** $EQS = (E, \Phi, e_0)$ consists of a set of equations E , a priority map Φ and an initial or start equation $e_0 \in E$, such that:

- Every term is constructed over atomic propositions.
 - Every $p \in \mathbb{P}$ is a term.
 - Every negated proposition $\neg p$ where $p \in \mathbb{P}$ is a term.

If t, t_1 and t_2 are terms, then the following are terms:

- $t_1 \vee t_2$
(Disjunction over terms)
- $t_1 \wedge t_2$
(Conjunction over terms)

– $\diamond t$
(Next operator over a term)

- $\Phi : T \rightarrow \mathbb{N}$ assigns a priority to each term.
- Every equation $e \in E$ is a pair consisting of a variable $v \in V$ and a term $t \in T$.
- The equation system has an initial equation e_0 .

Definition 30 An equation system is **closed**, if it contains an equation for every variable occurring in a term.

Note. We assume that all equation systems mentioned below are closed.

Definition 31 (Subterms) **Subterms** of the terms p , t_1 , t_2 and t are defined as follows:

- $Sub(p) = \{p\}$, $\forall p \in \mathbb{P}$
- $Sub(\neg p) = \{\neg p\}$, $\forall p \in \mathbb{P}$
- $Sub(t_1 \vee t_2) = \{t_1 \vee t_2\} \cup Sub(t_1) \cup Sub(t_2)$
- $Sub(t_1 \wedge t_2) = \{t_1 \wedge t_2\} \cup Sub(t_1) \cup Sub(t_2)$
- $Sub(\diamond t) = \{\diamond t\} \cup Sub(t)$

If T is a set of terms, we define

- $Sub(T) := \bigcup_{t \in T} Sub(t)$

2.5.2 Model Checking Games over Equation Systems

Section 2.4.4 defines and explains model checking games over μ TL formulas. In order to prove that equation systems and μ TL formulas are equi-expressive (see section 2.6), we need a similar technique for equation systems and thus decided to define model checking games over equation systems. The definition is directed towards model checking games over μ TL.

Definition 32 (Model checking games over EQS) Let $w \in \Sigma^\omega$ be a word and $EQS = (E, \Phi, e_0)$ be a closed equation system. Also, let T and V be the set of terms and variables occurring in the equations such that $e_j = (v_j, t_j)$ where $e_j \in E$, $v_j \in V$ and $t_j \in T$. A model checking game $G(v, EQS)$ between the two players \exists and \forall consists of

- a set of configurations $C = (w^{i..}, Sub(T))$
- an initial configuration $C_0 = (w, t_0)$
- the following set of rules:

$$(\vee) \frac{w^{i..}, e_1 \vee e_2}{w^{i..}, e_k} \quad (\exists), k \in \{1, 2\} \quad (\wedge) \frac{w^{i..}, e_1 \wedge e_2}{w^{i..}, e_k} \quad (\forall), k \in \{1, 2\}$$

$$(\diamond) \frac{w^{i..}, \diamond e}{w^{i+1..}, e} \quad (X) \frac{w^{i..}, X}{w^{i..}, e}, \text{ if } (X =_{\Phi(X)} e) \text{ is part of } EQS$$

If a configuration of the type $(w^{i..}, (e_1 \vee e_2))$ occurs, player \exists may choose between the disjuncts e_1 and e_2 . In situations of the sort $(w^{i..}, (e_1 \wedge e_2))$, it is \forall 's turn to choose. In all other cases, the next configuration is reached through a deterministic step according to the other rules.

- Winning conditions:

Player \exists wins a play if

- the play ends with $C_n = (w^{i..}, p)$ and $p \in w^{i..}$,
- the play ends with $C_n = (w^{i..}, \neg p)$ and $p \notin w^{i..}$ or
- the biggest priority occurring infinitely often is even.

Player \forall wins a play if

- the play ends with $C_n = (w^{i..}, p)$ and $p \notin w^{i..}$,
- the play ends with $C_n = (w^{i..}, \neg p)$ and $p \in w^{i..}$ or
- the biggest priority occurring infinitely often is odd.

Definition 33 (EQS Acceptance) The language of words accepted by an equation system E is defined as follows:

$$w \in L(E) :\Leftrightarrow \exists \text{ wins the model checking game } G(w, E).$$

Examples

This is the explanation of symbols used in the examples below:

$$EQS = \begin{pmatrix} x_0 =_{p_0} t_0 \\ \vdots \\ x_n =_{p_n} t_n \end{pmatrix}$$

The equation system EQS consists of the equations $(x_0 =_{p_0} t_0) \dots (x_n =_{p_n} t_n)$ where x_i is the variable of the respective equation, p_i the priority of x_i and t_i the term associated with x_i .

1. In this example, EQS_1 represents the atomic proposition a , EQS_2 represents b . EQS_3 , representing $(a \vee b)$ results from EQS_1 and EQS_2 and owns an additional third equation which describes the formula $(a \vee b)$. All priorities in EQS_3 are 0.

$$\begin{aligned} EQS_1 &= \begin{pmatrix} 0 =_0 a \end{pmatrix} \\ EQS_2 &= \begin{pmatrix} 0 =_0 b \end{pmatrix} \\ EQS_3 &= \begin{pmatrix} 0 =_0 a \\ 1 =_0 b \\ 2 =_0 a \vee b \end{pmatrix} \end{aligned}$$

2. Consider the following equation system:

$$EQS_4 = \begin{pmatrix} 0 =_0 a \\ 1 =_0 b \\ 2 =_1 b \vee (a \wedge (\diamond 2)) \\ 3 =_1 \diamond(b \vee (a \wedge (\diamond 2))) \end{pmatrix}$$

Emerging from EQS_1 and EQS_2 , which represent a and b , EQS_4 holds two supplementary equations. The third equation of EQS_4 represents $b \vee (a \wedge (\diamond 2))$. Finally, the last equation, adds a \diamond to the third equation which is tantamount to 'next'. Thus, EQS_4 contains the formula $\diamond(b \vee (a \wedge (\diamond 2)))$ and is the *initial equation*. The syntax tree for EQS_4 is shown in figure 2.8 and explains the priorities for each equation, see section 2.6.1.

2.6 Equivalence of μ TL Formulas and Equation Systems

This section will explain that the equation systems defined above and μ TL formulas are expressive by showing how they can be translated into each other. These processes justify the translation from PSL to μ TL via equation systems as mentioned in section 2.5.

2.6.1 Translating μ TL Formulas into Equation Systems

Definition 34 (*Syntax Tree*) A *syntax tree* of a formula φ is a tree representation of φ , in which all leaves represent either atomic propositions or variables whereas nodes are labelled with operators.

Example: The syntax tree for $\mu X.(b \vee (a \wedge \circ X))$ is shown in figure 2.8.

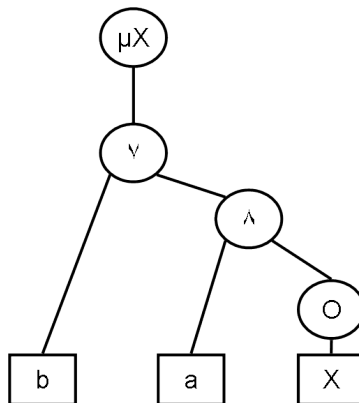


Figure 2.8: Example of a Syntax Tree

Definition 35 (*Priorities*) The **priority** of a formula is calculated from its syntax tree by using the following rules:

1. All leaves have got the priority 0.
2. If a node of the syntax tree contains \vee , \wedge or \circ , the priority of this node is the maximum priority of all nodes and leaves underneath.
3. If a node contains $\mu X.\varphi$, its priority depends on the priority of φ . If φ has an even priority, we add 1. If φ 's priority is odd, we keep the same priority for $\mu X.\varphi$.
4. If a node contains $\nu X.\varphi$, its priority depends again on the priority of φ . If φ has an odd priority, we add 1. If φ 's priority is even, we keep the same priority for $\nu X.\varphi$.
5. Every variable has the priority $\Omega(X)$, which is equal to the priority of the corresponding $\sigma X.\varphi$.

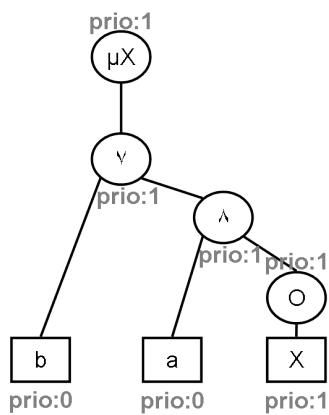


Figure 2.9: Priorities of a Syntax Tree

Example: The priorities for the example given above are shown in figure 2.9.

Definition 36 (Equation systems over μTL formulas) Let $\varphi \in \mu TL$ be closed. Define $EQS_\varphi = (E_\varphi, \Phi_\varphi, e_{0_\varphi})$ and let the variables of EQS_φ be $\{V_\psi \mid \psi \in Sub(\varphi)\}$. The equation system EQS_φ over φ can be build as follows:

- For all $\psi \in Sub(\varphi)$, EQS_φ contains an equation $V_\psi =_{\Omega(\psi)} t_\psi$, such that

$$t_\psi = \begin{cases} a & , \text{ if } \psi = a \\ V_{\psi_1} \vee V_{\psi_2} & , \text{ if } \psi = \psi_1 \vee \psi_2 \\ V_{\psi_1} \wedge V_{\psi_2} & , \text{ if } \psi = \psi_1 \wedge \psi_2 \\ \diamond V_{\psi_1} & , \text{ if } \psi = \bigcirc \psi_1 \\ V_\psi & , \text{ if the variable } Y \text{ is defined by } \sigma Y.\psi \end{cases}$$

and $t_{\sigma Y.\psi} = t_Y$.

- The equation system E_φ consists of all equations translated according to the map above:
 $E_\varphi = \{(V_\psi, t_\psi) \mid \psi \in Sub(\varphi)\}$
- The priorities of each equation are explained in definition 35.
- The initial equation $e_{0_\varphi} = (V_\varphi, t_\varphi)$ is the equation containing all translated subformulas of φ .

Lemma 5 For all $w \in \Sigma$, if φ is a closed μ TL formula and EQS_φ the equation system obtained by translating φ as described above, the following holds:

$$w \vDash_{\mu TL} \varphi \Leftrightarrow \exists \text{ wins the model checking game over } w \text{ and } EQS_\varphi.$$

Proof (outline)

Suppose that $w \vDash_{\mu TL} \varphi$. This is equivalent to saying that \exists has a winning strategy ζ for the model checking game $G = (w, \varphi)$. Now, we define the strategy $\tilde{\zeta}$ which would be used by \exists when playing the game $\tilde{G} = (w, EQS_\varphi)$:

$$\tilde{\zeta}(w^{i..}, t_\psi) = \zeta(w^{i..}, \psi)$$

It is easy to see that $\tilde{\zeta}$ a winning strategy for \exists if ζ is: Imagine that \forall wins the game \tilde{G} , which means that, for an infinite play, the biggest priority occurring infinitely often is odd. This priority emanates from translating a formula or subformula whose biggest variable with respect to $<_\varphi$ is μ . Thus, \exists would not have won the game G by using ζ , which leads to a contradiction.

Analogously, if $\tilde{\zeta}$ is a winning strategy for \exists , if ζ is.

q.e.d.

Example

Consider the μ TL formula $\varphi = \nu X. \mu Y. (\neg p \wedge \bigcirc X) \vee (p \wedge \bigcirc(p \wedge \bigcirc Y))$. The subformulas of φ are:

- $\nu X. \mu Y. (\neg p \wedge \bigcirc X) \vee (p \wedge \bigcirc(p \wedge \bigcirc Y))$
- $\mu Y. (\neg p \wedge \bigcirc X) \vee (p \wedge \bigcirc(p \wedge \bigcirc Y))$
- $(\neg p \wedge \bigcirc X) \vee (p \wedge \bigcirc(p \wedge \bigcirc Y))$
- $(\neg p \wedge \bigcirc X)$
- $\neg p$
- $\bigcirc X$
- X
- $(p \wedge \bigcirc(p \wedge \bigcirc Y))$

- p
- $\circ(p \wedge \circ Y)$
- $p \wedge \circ Y$
- $\circ Y$
- Y

Now, they can be converted into equations. For a better overview, we will start with the shortest subformulas and thus change order of the subformulas listed above. Also, variables have been renamed. The calculation of priorities is explained in section 2.5.

$$EQS_{\varphi} = \left(\begin{array}{l} 0 =_0 p \\ 1 =_0 \neg p \\ 2 =_0 (\neg p \wedge \diamond 7) \\ 3 =_0 (p \wedge \diamond 6) \\ 4 =_0 (p \wedge (p \wedge \diamond 6)) \\ 5 =_0 (\neg p \wedge \diamond 7) \vee (p \wedge (p \wedge \diamond 6)) \\ 6 =_1 5 \\ 7 =_2 6 \end{array} \right)$$

The initial equation is the one for variable 7; it represents φ .

2.6.2 Translating Equation Systems into μ TL Formulas

Let $EQS = (E, \Phi, e_0)$ be an equation system. EQS can be translated into a μ TL formula through the following steps:

- Introduce a new initial equation e_0 which owns a new variable but the same term as e_0 . Assign a priority higher than all other priorities in E to e_0 .
- Sort all equations in decending order according to their priorities:

$$EQS = \left(\begin{array}{l} x_0 =_{p_0} t_0 \\ \quad \quad \quad \cdot \cdot \\ x_n =_{p_n} t_n \end{array} \right),$$

such that $p_0 \geq p_1 \geq \dots \geq p_n$.

- Translate every equation $x_i =_{p_i} t_i$ into $\sigma X_i. \|t_i\|$ where $\sigma = \begin{cases} \mu & , \text{ if } p_i \text{ is odd} \\ \nu & , \text{ else.} \end{cases}$

and

- $\|p\| := p$ for atomic propositions $p \in \mathbb{P}$
- $\|\varphi_1 \wedge \varphi_2\| := \|\varphi_1\| \wedge \|\varphi_2\|$
- $\|\varphi_1 \vee \varphi_2\| := \|\varphi_1\| \vee \|\varphi_2\|$
- $\|\odot\varphi\| := \diamond\|\varphi\|$
- $\|X_j\| := \begin{cases} X_j & , \text{ if } j \leq i \\ \sigma X_j. \|t_j\| & \text{ else.} \end{cases}$

After this process, $\sigma X_0. \|t_0\|$ is equivalent to the equation system.

Lemma 6 *Let w be a word in Σ^ω , EQS be a closed equation system and $\psi_{EQS} := \sigma X_0. \|t_0\|$ the formula obtained by translating EQS by using the process defined above.*

$$\exists \text{ wins the model checking game } G = (w, EQS) \Leftrightarrow w \models_{muTL} \psi_{EQS}$$

Proof (outline)

Suppose that \exists wins the model checking game $G = (w, EQS)$ by using the strategy ζ . Now define the strategy $\tilde{\zeta}$ which will be used by \exists when playing the game $\tilde{G} = (w, \psi_{EQS})$:

$$\tilde{\zeta}(w^{i..}, \varphi) := \zeta(w^{i..}, t_\varphi)$$

If \forall wins \tilde{G} , the biggest variable occurring infinitely often in ψ_{EQS} with respect to $<_\varphi$ is μ . As this means that the biggest priority occurring in EQS must be odd, \exists would not win G , which is a contradiction to the initial supposition.

Analogously, if $\tilde{\zeta}$ is a winning strategy for \exists , if ζ is.

q.e.d.

Example

We will now translate the equation system

$$EQS = \left(\begin{array}{l} 0 =_0 p_1 \\ 1 =_0 p_2 \\ 2 =_0 (p_1 \vee p_2) \wedge \diamond 2 \end{array} \right)$$

with the initial equation 2. First, we introduce a new initial equation with a higher priority than all other equations have:

$$3 =_1 (p_1 \vee p_2) \wedge \diamond 2$$

Now, we have to sort the equations ...

$$EQS = \begin{pmatrix} 0 =_1 p_1 \vee (p_2 \wedge \diamond 3) \\ 1 =_0 p_1 \\ 2 =_0 p_2 \\ 3 =_0 (p_1 \vee p_2) \wedge \diamond 3 \end{pmatrix}$$

... and translate them:

- $\psi_0 = \mu X_0.(p_1 \vee p_2) \wedge \circ \psi_3$
- $\psi_1 = \nu X_1.(p_1)$
- $\psi_2 = \nu X_2.(p_2)$
- $\psi_3 = \nu X_3.(p_1 \vee p_2) \wedge \circ X_3$

We can now paste ψ_3 into ψ_0 and receive the μ TL formula represented by EQS :

- $\psi_0 = \mu X_0.(p_1 \vee p_2) \wedge \circ (\nu X_3.(p_1 \vee p_2) \wedge \circ X_3)$

which is equivalent to

- $\psi_0 = \nu X_3.(p_1 \vee p_2) \wedge \circ X_3.$

This formula has also been explained in section 2.4.3.

3 Translating PSL to μ TL

This chapter will prove that every PSL formula can be translated into an equivalent μ TL formula. For that purpose, a few supplementary definitions and theorems are needed and will be explained just before the main proof.

Note. All functions that translate some formula or automaton will be described by $tr_{X \rightarrow Y}$, meaning that the formula or automaton of type X is translated into type Y .

3.1 Preparations

3.1.1 Automata Constructions

Definition 37 (*Translating boolean expressions to NFAs*) If b is a boolean expression, let $A = (S, \Sigma, \delta, s_0, F)$ be the NFA A such that

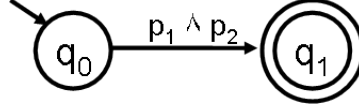
- $S = \{q_0, q_1\}$
- $\delta(q_0) = \{(b, q_1)\}, \delta(q_1) = \emptyset$
- $s_0 = q_0$
- $F = \{q_1\}$

The translation is called $tr_{\text{BOOL} \rightarrow \text{NFA}}$.

Lemma 7 If $A := tr_{\text{BOOL} \rightarrow \text{NFA}}(b)$ was obtained from transforming a boolean expression b , A accepts a word $w \in \Sigma^*$ if $|w| = 1$ and $w^0 \models b$.

Proof: Can easily be done by induction on the structure of boolean expressions.

Example: Let b be a boolean expression, p_1 as well as p_2 atomic propositions and $b = p_1 \wedge p_2$. The automaton $A = tr_{\text{BOOL} \rightarrow \text{NFA}}(b)$ is shown in figure 3.1.

Figure 3.1: NFA obtained from translating $p_1 \wedge p_2$

Definition 38 (ϵ -NFA) We call $A_\epsilon = (S, \Sigma, \delta, s_0, F)$ an ϵ -NFA, if

- $S = \{q_0\}$
- $\delta(q_0) = \emptyset$
- $s_0 = q_0$
- $F = q_0$

Lemma 8 The ϵ -NFA accepts the empty word ϵ .

Proof: Immediate.

Definition 39 (Translating regular expressions to NFAs) Let r_1, r_2 and r be regular expressions and b be a boolean expression. Using the definitions 8, 9, 10, 11, 37 and 38, every r can be translated into an NFA $A := tr_{REG \rightarrow NFA}(r)$ by using the following functions:

$$tr_{REG \rightarrow NFA}(r) = \begin{cases} tr_{BOOL \rightarrow NFA}(b) & , \text{ if } r = b \\ A_\epsilon & , \text{ if } r = \epsilon \\ (tr_{REG \rightarrow NFA}(r_1)); (tr_{REG \rightarrow NFA}(r_2)) & , \text{ if } r = r_1; r_2 \\ (tr_{REG \rightarrow NFA}(r_1)) \cup (tr_{REG \rightarrow NFA}(r_2)) & , \text{ if } r = r_1 \cup r_2 \\ (tr_{REG \rightarrow NFA}(r_1)) \cap (tr_{REG \rightarrow NFA}(r_2)) & , \text{ if } r = r_1 \cap r_2 \\ (tr_{REG \rightarrow NFA}(r_1))^* & , \text{ if } r = r_1^* \end{cases}$$

Lemma 9 If $w \in \Sigma^\omega$ is a word and $A := tr_{REG \rightarrow NFA}(r)$ was obtained from translating the regular expression r , the following holds: $w \in L(r) \Leftrightarrow w \in L(A)$.

Proof: Combine the lemmas 1, 2, 3, 4, 7 and 8.

Definition 40 (Translating NFAs into μ TL formulas) Let φ be a closed μ TL formula, let $\{q_0, \dots, q_n\}$ be the states of an NFA A and q_0 be the initial state. In order to translate A into μ TL, we define a μ TL formula ψ_i for every state such that

$$\psi_i = \mu X_i. \bigvee_{(g,q_j) \in \delta(q_i)} g \wedge (\alpha_j \vee \bigcirc \chi_{i_j})$$

such that

$$\chi_{i_j} = \begin{cases} X_j & , \text{ if } j \leq i \\ \psi_j & \text{ otherwise.} \end{cases}$$

and

$$\alpha_j = \begin{cases} \varphi & , \text{ if } q_j \in F \\ \perp & \text{ otherwise.} \end{cases}$$

We name this translation $tr_{NFA \times \mu TL \rightarrow \mu TL}(A)$.

Lemma 10 *Let A be an NFA, φ be closed a μ TL formula and w be a word in Σ^ω .*

$$w \models tr_{NFA \times \mu TL \rightarrow \mu TL}(A) \text{ iff there is an } i \text{ such that } w^{0..i} \in L(A) \text{ and } w^{i..} \models_{\mu TL} \varphi.$$

Proof This lemma will be proved by showing that \exists has a winning strategy in the model checking game $G(w, tr_{NFA \rightarrow \mu TL}(A))$ iff there is an i such that A accepts $w^{0..i}$ and $w^{i..} \models_{\mu TL} \varphi$.

Note that if \exists reaches a configuration of the type (w, φ) , one can conclude that $w \models_{\mu TL} \varphi$, because φ is a closed formula by assumption, so that nothing depends on what happened before. Furthermore, if \exists has a winning strategy at any configuration c_i of a play and performs a choice which is part of that strategy and leads to a configuration c_j , \exists also has a winning strategy at configuration c_j .

” \Leftarrow ”:

By assumption, there is an $i \in \mathbb{N}$ and an accepting run of A over the prefix $w^{0..i}$ of w . Let $\psi_0 = tr_{NFA \rightarrow \mu TL}(A)$ be the formula obtained from translating A as described in definition 40. Note that ψ_0 only contains μ quantifiers, so that \exists has to make the right choices at disjunctions in ψ_0 and every play has to end after a finite number of steps.

For every state of A , there is a μ TL formula of the form

$$\psi_i = \mu X_i. \bigvee_{(g,q_j) \in \delta(q_i)} g \wedge (\alpha_j \vee \bigcirc \chi_{i_j})$$

\exists has to choose between disjuncts in two different situations:

1. In the first situation, namely $\mu X_i. \bigvee_{(g,q_j) \in \delta(q_i)} \dots$, \exists wants to make her choices depending on the accepting run of A over w : if the run contains a transition from q_i to q_j under the symbol w^i using guard g , \exists chooses the disjunct $g \wedge (\alpha_j \vee \bigcirc \chi_{i_j})$ in ψ_i .
Now, \forall may choose between the conjuncts g and $(\alpha_j \vee \bigcirc \chi_{i_j})$ in ψ_i . As \exists chose $g \wedge (\alpha_j \vee \bigcirc \chi_{i_j})$ only if the run uses the respective transition, $w^i \vDash_{\mu TL} g$. Thus, \forall will choose $(\alpha_j \vee \bigcirc \chi_{i_j})$ which leads to the following:
2. In situations of the second type, like $(\alpha_j \vee \bigcirc \chi_{i_j})$, \exists chooses α_j , if q_j is an accepting state and A has reached the end of the prefix $w^{0..i}$ when moving to q_j .

As the prefix $w^{0..i}$ is finite, \exists will choose φ after a finite number of steps. In that way, \exists has reached a configuration (v, φ) such that $v = w^{i..}$. By hypothesis, $w^{i..} \vDash_{\mu TL} \varphi$ so that \exists will also win the game over v and φ . \checkmark

” \Rightarrow ”:

Suppose that \exists wins the model checking game $G(w, tr_{NFA \times \mu TL \rightarrow \mu TL}(A))$. Now, we have to find a way to decompose w into a prefix $w^{0..k}$ and a suffix $w^{k..}$ such that A accepts $w^{0..k}$ and $w^{k..} \vDash_{\mu TL} \varphi$. Suppose that \forall chooses his best strategy. A strategy s_1 is better than a strategy s_2 , if the results that a player achieves with this strategy are better than the results that can be achieved with s_2 . If \forall plays his best strategy against \exists 's winning strategy, we are in a well-defined play. As \exists wins the play, she has to choose φ in the formulas $\psi_i = \mu X_i. \bigvee_{(g,q_j) \in \delta(q_i)} g \wedge (\alpha_j \vee \bigcirc \chi_{i_j})$ (where $\alpha_j = \varphi$, if $q_j \in F$) at some configuration of the play because she will certainly not choose \perp . Otherwise, the play would be infinite and the outermost variable with respect to $<_{\varphi}$ would be μ which is a contradiction to the assumption that \exists wins G .

This means that \exists reaches a configuration (v, φ) such that $v = w^{k..}$ and therefore, $v \vDash_{\mu TL} \varphi$. Also, by choosing the transitions in A that \exists would have chosen in the corresponding $\mu X_i. \bigvee_{(g,q_j) \in \delta(q_i)} \dots$, we can find an accepting run of A over w . Altogether, $w^{0..k}$ is accepted by A and $w^{k..} \vDash_{\mu TL} \varphi$.

q.e.d.

Example: Let A be the automaton presented in figure 3.1 and φ be a μ TL formula. For the states q_0 and q_1 , we define

- $\psi_0 = \mu X_0.(p_1 \wedge p_2) \wedge (\varphi \vee \bigcirc \psi_1)$
- $\psi_1 = \mu X_1. \bigcirc X_1$

and insert ψ_1 into ψ_0 :

- $\psi_0 = \mu X_0.(p_1 \wedge p_2) \wedge (\varphi \vee \circ(\mu X_1. \circ X_1))$
 $\Leftrightarrow \psi_0 = \mu X_0.(p_1 \wedge p_2) \wedge (\varphi)$
 $\Leftrightarrow \psi_0 = (p_1 \wedge p_2) \wedge (\varphi)$

For all words w , $w \vDash_{\mu TL} tr_{NFA \rightarrow \mu TL}(A)$ iff $w^0 \vDash_{\mu TL} p_1 \wedge p_2$ and $w^{0..} \vDash_{\mu TL} \varphi$.

Definition 41 (Translating NFAs to NBAs) Let $A = (S, \Sigma, \delta, s_0, F)$ be an NFA. The NBA $\tilde{A} = (\tilde{S}, \Sigma, \tilde{\delta}, s_0, F)$ is obtained from A by the following changes:

- Let $S_0 \subseteq S$ be the set of all states from which no final state is reachable. $\tilde{S} := S \setminus S_0$.
- For all states $q \in \tilde{S}$, we define the transition function

$$\tilde{\delta}(q) = \begin{cases} \delta(q) \upharpoonright_{\tilde{S}} \cup \{(true, q)\} & , \text{ if } q \in F \\ \delta(q) \upharpoonright_{\tilde{S}} & \text{ otherwise.} \end{cases}$$

where $\delta(q) \upharpoonright_{\tilde{S}} := \{(g, q') \mid q' \in \tilde{S} \text{ and } (g, q') \in \delta(q)\}$.

We name this translation $tr_{NFA \rightarrow NBA}$.

Lemma 11 Let $A = (S, \Sigma, \delta, s_0, F)$ be an NFA and $\tilde{A} = tr_{NFA \rightarrow NBA}(A)$, $\tilde{A} = (\tilde{S}, \Sigma, \tilde{\delta}, s_0, F)$ be the corresponding Büchi automaton (see definition 41). The following holds:

$$L(\tilde{A}) = \{ w \in \Sigma^\omega \mid (\exists \text{ a finite prefix } u \preceq w, \text{ such that } u \in L(A)) \\ \text{ or } (\forall \text{ finite prefixes } u \preceq w : \exists v \in \Sigma^*, \text{ s.t. } uv \in L(A)) \}$$

Proof:

- First we show that $L(\tilde{A})$ is a subset of the set of words on the right hand side of the equation. Suppose that $w \in \Sigma^\omega$ and let $\tilde{\rho} = q_0 q_1 \dots$ be an accepting run of \tilde{A} over w . This means that an accepting state $q_i \in \tilde{F}$ of \tilde{A} has been reached infinitely often. There are two possibilities:
 - $\tilde{\rho}$ contains an accepting state q_i which was also final in A : $q_i \in F$. This means we can create an accepting run by staying in this state using the transition $(true, q_i) \in \delta(q_i)$. W.l.o.g. let q_i be the first accepting state in $\tilde{\rho}$. We can now conclude that we have only traversed transitions that are also part of A prior to reaching q_i and thus, A accepts a prefix of w . \checkmark

- $\tilde{\rho}$ does not contain an accepting state $q_i \in F$, but we will only pass through states from which we could still reach an accepting state. This means that we can extend w and all its prefixes $w' \leq w$ with a suffix v such that $w'v$ would be accepted by A .
✓
- Now we show that the set of words specified above is a superset of $L(A)$. There are again two cases:
 - Let u be a prefix of w which is in $L(A)$. Thus, there is an accepting run of A over u which obviously only contains states that can reach a final state. This means that the states of this run are also part of \tilde{A} . From a state that is final in A , every word will be accepted in \tilde{A} because of the supplementary transitions $(true, q) \in \delta(q)$ if $q \in F$. Thus, \tilde{A} accepts w . ✓
 - Suppose that all finite prefixes $u_i \leq w$, $i \in \mathbb{N}$, can be continued by a word $v_i \in \Sigma^*$ so that $u_i v_i$ is accepted by A .

Let ρ_i be the respective accepting run of A over $u_i v_i$ and let $\dot{\rho}_i$ be the prefix of ρ_i which originates from reading u_i without the respective v_i . Note that $\dot{\rho}_i$ consists of $i + 1$ states. (u_0 is the empty word, so $\dot{\rho}_1$ has length 1, and so on.)

If Q is the set of states in A , let $GR(V, E)$ be a graph consisting of the vertices $V = Q \times \mathbb{N}$ and the edges E , being generated by all runs $\dot{\rho}_i$, such that

- * $(q, i) \mapsto (q', i + 1)$ if $\exists j \geq i$ such that $\dot{\rho}_j$ contains the transition from q to q' under some guard g such that $u^i \models g$ at position i of the run, for $w^i = a$

We can state that

- * GR is a directed acyclic graph and its initial vertex is $(q_0, 0)$ if q_0 is the initial state of A or \tilde{A} , respectively.
- * As there is a partial run $\dot{\rho}_k$ with length $k + 1$ for every $k \in \mathbb{N}$, GR contains paths of arbitrary length.
- * GR is finitely branching because there cannot be more edges of the form $(q, i) \mapsto (q', i + 1)$ than there are states in A .

Based on König's lemma [Kön36], there is an infinite path in GR . We are now going to show that this path is an accepting run of \tilde{A} over w .

First, notice that the runs $\dot{\rho}_i$ only pass through states that are also part of \tilde{A} because those runs can be extended to an accepting run ρ_i . This means that all these states can reach a final state in A and \tilde{A} . Beyond that, we observe that if there is an edge $(q, i) \mapsto (q', i + 1)$ in GR , there must be a transition $(g, q') \in \tilde{\delta}(q)$ such that $w^i \models g$. The reason for this is that \tilde{A} contains equally many or more transitions in states

that are part of A and \tilde{A} .

Thus, we have found an infinite path of GR which is an accepting run of \tilde{A} over w , because all states in \tilde{A} are final states. \checkmark

q.e.d.

Example: Let p_1 and p_2 be atomic propositions and $r = p_1; p_2$. Thus, if w is a word, $w \in L(r)$ iff $w^0 \models_{PSL} p_1$ and $w^1 \models_{PSL} p_2$. The NFA $A = tr_{REG \rightarrow NFA}(r)$, accepting words that are in $L(r)$, is shown in figure 3.2.

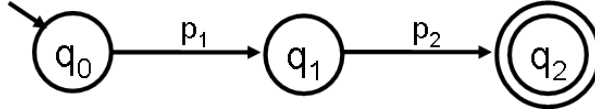


Figure 3.2: NFA $A = tr_{REG \rightarrow NFA}(r)$

As there are no states which cannot reach the final state q_2 , we only add the transition ($true, q_2$) to q_2 in order to receive $\tilde{A} = tr_{NFA \rightarrow NBA}(A)$, see figure 3.3.

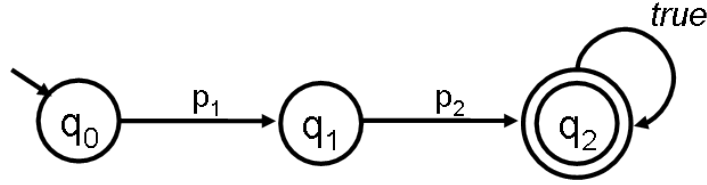


Figure 3.3: NBA $\tilde{A} = tr_{NFA \rightarrow NBA}(A)$

For the words $w_1 = \begin{matrix} p_1 & | & 1 & 0 & 0 \dots \\ p_2 & | & 0 & 1 & 0 \dots \end{matrix}$, $w_2 = \begin{matrix} p_1 & | & 1 & 1 & 1 \dots \\ p_2 & | & 1 & 1 & 1 \dots \end{matrix}$ and $w_3 = \begin{matrix} p_1 & | & 1 & 1 & 1^\omega \\ p_2 & | & 0 & 0 & 0^\omega \end{matrix}$, we can now see that:

- $w_1 \models_{PSL} r$ and $w_2 \models_{PSL} r$ because the words $w_1^0 \top^\omega$ and $w_1^1 \top^\omega$ - as well as $w_2^0 \top^\omega$ and $w_2^1 \top^\omega$ - all satisfy $r \diamond \rightarrow true$. \tilde{A} accepts w_1 and w_2 because it has reached and stays in the final state after reading w_1^1 or w_2^1 , respectively.
- $w_3 \not\models_{PSL} r$: the suffix $w_3^{0..1} \not\models_{PSL} r \diamond \rightarrow true$ because at position w_3^1 , p_2 is not fulfilled. Also, \tilde{A} does not accept w_3 , because it gets stuck in q_1 when reading w_3^1 and will not reach the final state.

Definition 42 (Trivial NBAs) Let A be an NBA. We say that A is *trivial*, if all states of A are final.

Definition 43 (Translating trivial NBAs into μ TL formulas) Let φ be a closed μ TL formula and $A = (S, \Sigma, \delta, s_0, S)$ be a trivial NBA such that $S = \{q_0, \dots, q_n\}$ and $s_0 = q_0$. For every state q_i , we define a μ TL formula ψ_i of the form

$$\psi_i = \nu X_i. \bigvee_{(g, q_j) \in \delta(q_i)} g \wedge \bigcirc \chi_{i_j}$$

such that

$$\chi_{i_j} = \begin{cases} X_j & , \text{ if } j \leq i \\ \psi_j & \text{ otherwise.} \end{cases}$$

We call this translation $tr_{NBA \rightarrow \mu TL}$.

Lemma 12 If w is a word in Σ^ω and A a trivial NBA, the following holds:

$$w \models_{\mu TL} tr_{NBA \rightarrow \mu TL}(A) \Leftrightarrow w \in L(A)$$

Proof We show the correctness of this lemma demonstrating that \exists has a winning strategy in the model checking game $G(w, tr_{NBA \rightarrow \mu TL}(A))$ iff A accepts w .

” \Leftarrow ”:

By hypothesis, there is an accepting run of A over w . This means that there is an infinite run $\rho = q_0 q_1 \dots$ of A over w . We need to find a corresponding strategy for \exists and show that it is a winning strategy.

Let ψ_0 be the formula obtained from translating A , see definition 43. Note that ψ_0 only contains ν quantifiers.

For every state of A , we have a μ TL formula of the form

$$\psi_i = \nu X_i. \bigvee_{(g, q_j) \in \delta(q_i)} g \wedge \bigcirc \chi_{i_j}$$

\exists may perform her choices depending on ρ : if ρ contains the transition $(g, q_j) \in \delta(q_i)$, \exists chooses the corresponding disjunct in configurations of the type $(w^i, (g \wedge \bigcirc \chi_{i_j}) \vee \dots)$. In configurations that arise from subformulas of some g , \exists wants to perform her choices with respect to the winning strategies for games of the type $G(w, g)$ which exist because her previous choices induce that $w^i \models g$.

If \forall always chooses $\bigcirc \chi_{i_j}$, this leads to an infinite play, \exists wins because ψ_0 only contains ν

quantifiers.

On the other hand, if the play is not finite, \forall must have chosen g in some ψ_i . As g is fulfilled by hypothesis, \exists wins again. \checkmark

" \Rightarrow ":

Suppose that \exists has a winning strategy in the game $G(w, tr_{NBA \rightarrow \mu TL}(A))$ and \forall plays according to his best strategy so that we are in a well-defined play.

All formulas in $tr_{NBA \rightarrow \mu TL}(A)$ are of the type

$$\nu X_i. \bigvee_{(g,q_j) \in \delta(q_i)} g \wedge \bigcirc \chi_{i_j}$$

In this play, \forall 's best choice will always be to choose the conjunct on the right hand side of a formula of the type $(g \wedge \bigcirc \chi_{i_j})$ because as \exists wins, she will never have chosen a disjunct $(g \wedge \bigcirc \chi_{i_j})$ where g cannot be fulfilled. We can conclude that $w^i \models_{\mu TL} g$ for all i and the respective guard g , see theorem 1.

Thus, \forall always chooses $(\bigcirc \chi_{i_j})$ and we are obviously in an infinite play.

Now, we can construct an infinite run of A over w by passing through the transitions $(g, q_j) \in \delta(q_i)$ such that \exists chooses the disjunct $g \wedge \bigcirc q_j$. As all states of A are final and we have found an infinite run of A over w , A accepts w . \checkmark

q.e.d.

3.1.2 Properties of μ TL and PSL Formulas

Lemma 13 *Let φ and ψ be PSL formulas and $\tilde{\varphi}$ and $\tilde{\psi}$ be μ TL formulas such that, for any word $w \in \Sigma^\omega$, $w \models_{PSL} \varphi \Leftrightarrow w \models_{\mu TL} \tilde{\varphi}$ and $w \models_{PSL} \psi \Leftrightarrow w \models_{\mu TL} \tilde{\psi}$. The following holds:*

$$w \models_{PSL} \varphi U \psi \Leftrightarrow w \models_{\mu TL} \mu X. (\tilde{\psi} \vee (\tilde{\varphi} \wedge \bigcirc X))$$

Note. This lemma will not be proved here because the fact that the until-operator can be modelled using the least fixed point is part of the linear time μ -calculus standard.

Definition 44 (Negation of μTL formulas) Let φ be a μTL formula. The notation $\varphi[\neg X/X]$ means that every occurrence of $\neg X$ in φ is replaced with X . The negated μTL formula $\neg\varphi$ is defined as:

$$\neg\psi := \begin{cases} \neg p & , \text{ if } \psi = p, p \in \mathbb{P} \\ p & , \text{ if } \psi = \neg p, p \in \mathbb{P} \\ X & , \text{ if } \psi = \neg X \\ (\neg\varphi_1) \wedge (\neg\varphi_2) & , \text{ if } \psi = \neg(\varphi_1 \vee \varphi_2) \\ (\neg\varphi_1) \vee (\neg\varphi_2) & , \text{ if } \psi = \neg(\varphi_1 \wedge \varphi_2) \\ \bigcirc(\neg\varphi) & , \text{ if } \psi = \neg(\bigcirc\varphi) \\ \nu X. \neg(\varphi[\neg X/X]) & , \text{ if } \psi = \neg(\mu X. \varphi) \\ \mu X. \neg(\varphi[\neg X/X]) & , \text{ if } \psi = \neg(\nu X. \varphi) \end{cases}$$

Lemma 14 For every μTL formula φ , there is a closed formula $\neg\varphi$ such that for any word w , $w \models_{\mu\text{TL}} \neg\varphi \Leftrightarrow \bar{w} \not\models_{\mu\text{TL}} \varphi$.

The proof can easily be done by induction on the structure of μTL formulas and is explained, for example, in [Dax06].

Lemma 15 For all words w in Σ^ω and all regular expressions r the following holds:

$$w \models_{\text{PSL}} r \Leftrightarrow (\exists \text{ a finite prefix } u \leq w \text{ s.t. } u \models_{\text{PSL}} r) \\ \text{or } (\forall \text{ finite prefixes } u \leq w \text{ there is a } v \text{ s.t. } uv \models_{\text{PSL}} r)$$

Proof.

” \Rightarrow ”:

Suppose that $w \models_{\text{PSL}} r$. By definition 18, $w \models_{\text{PSL}} r \Leftrightarrow \forall \text{ finite } u \leq w : u \top^\omega \models_{\text{PSL}} r \diamond \rightarrow \text{true}$. We can distinguish between two cases:

- There is a some prefix $w^{0..i}$ such that $w^{0..i} \models_{\text{PSL}} r$ and $w^{i..} \not\models_{\text{PSL}} r$. Obviously, $w^{0..i} \in L(r)$. \checkmark
- If no such prefix $w^{0..i} \in L(r)$ can be found but for all prefixes $u \leq w$, $u \top^\omega \models_{\text{PSL}} r \diamond \rightarrow \text{true}$, there is some \tilde{w} such that $w \leq \tilde{w}$ and $\tilde{w} \models_{\text{PSL}} r \diamond \rightarrow \text{true}$. Thus, every finite $u \leq w$ can be continued by a suffix v such that $uv \models_{\text{PSL}} r$. \checkmark

" \Leftarrow ":

- Suppose that there is a finite prefix $u \leq w$ such that $u \models_{PSL} r$. This means that all words $u_i \leq u$ can be continued by \top^ω so that $u_i \top^\omega \models_{PSL} r \diamond \rightarrow true$, because replacing letters of u with \top does not change the fact that $u \in L(r)$ and after reading u , $true$ will be fulfilled by \top^ω . Also, all words $u_j \top^\omega$ such that $u \leq u_j$ are in $L(r)$, because after reading u , the condition " $true$ " will always be fulfilled and therefore, $w \models_{PSL} r$. \checkmark
- If there is no finite prefix of w which lies in $L(r)$, we require that every $u \leq w$ can be continued with a v such that $uv \models_{PSL} r$. This implies in particular that every $u \leq w$ can be continued with $\tilde{v} := \top \top \top \dots \top$ so that $u\tilde{v} \models r$ and obviously, the suffix \top^ω will fulfill the condition " $true$ ". Thus, $w \models_{PSL} r$. \checkmark

3.2 Translation and Proof

Definition 45 Let b be a boolean expression, r be a regular expression and φ_1, φ_2 as well as φ and ψ be PSL formulas. We define the translation of PSL formulas to μ TL formulas as follows:

$$tr_{PSL \rightarrow \mu TL}(\psi) = \begin{cases} b & , \text{ if } \psi = (b!) \\ \neg(tr_{PSL \rightarrow \mu TL}(\varphi)) & , \text{ if } \psi = (\neg\varphi) \\ tr_{PSL \rightarrow \mu TL}(\varphi_1) \vee tr_{PSL \rightarrow \mu TL}(\varphi_2) & , \text{ if } \psi = (\varphi_1 \vee \varphi_2) \\ \bigcirc tr_{PSL \rightarrow \mu TL}(\varphi) & , \text{ if } \psi = (X!\varphi) \\ \mu X.(tr_{PSL \rightarrow \mu TL}(\varphi_2) \vee (tr_{PSL \rightarrow \mu TL}(\varphi_1) \wedge \bigcirc X)) & , \text{ if } \psi = (\varphi_1 U \varphi_2) \\ tr_{NFA \times \mu TL \rightarrow \mu TL}(tr_{REG \rightarrow NFA}(r), \varphi) & , \text{ if } \psi = (r \diamond \rightarrow \varphi) \\ tr_{NBA \rightarrow \mu TL}(tr_{NFA \rightarrow NBA}(tr_{REG \rightarrow NFA}(r))) & , \text{ if } \psi = (r) \end{cases}$$

Theorem 2 (Equivalence of μ TL and PSL) If v is a word in Σ^ω and ψ is a PSL formula, the following holds:

$$v \models_{PSL} \psi \Leftrightarrow v \models_{\mu TL} tr_{PSL \rightarrow \mu TL}(\psi)$$

Proof: The proof is done by induction on the structure of PSL formulas.

1. $\psi = (b!)$:
 - $v \models_{PSL} b!$
 - $\Leftrightarrow v^0 \models b$ (according to definition 18)
 - $\Leftrightarrow v \models_{\mu TL} b$ (by induction on the structure of b)
 - $\Leftrightarrow v \models_{\mu TL} tr_{PSL \rightarrow \mu TL}(b!)$ (according to definition 45)

2. $\psi = (\neg\varphi)$:

$$\begin{aligned}
& v \vDash_{PSL} \neg\varphi \\
& \Leftrightarrow \bar{v} \vDash_{PSL} \varphi \text{ (according to definition 18)} \\
& \Leftrightarrow \bar{v} \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\varphi) \text{ (by hypothesis)} \\
& \Leftrightarrow v \vDash_{\mu TL} \neg(tr_{PSL \rightarrow \mu TL}(\varphi)) \text{ (by hypothesis)} \\
& \Leftrightarrow v \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\neg\varphi) \text{ (according to definition 45)}
\end{aligned}$$

3. $\psi = (\varphi_1 \vee \varphi_2)$:

$$\begin{aligned}
& v \vDash_{PSL} \varphi_1 \vee \varphi_2 \\
& \Leftrightarrow v \vDash_{PSL} \varphi_1 \text{ OR } v \vDash_{PSL} \varphi_2 \text{ (according to definition 18)} \\
& \Leftrightarrow v \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\varphi_1) \text{ OR } v \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\varphi_2) \text{ (by hypothesis)} \\
& \Leftrightarrow v \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\varphi_1) \vee tr_{PSL \rightarrow \mu TL}(\varphi_2) \text{ (according to definition 24)} \\
& \Leftrightarrow v \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\varphi_1 \vee \varphi_2) \text{ (according to definition 45)}
\end{aligned}$$

4. $\psi = (X!\varphi)$:

$$\begin{aligned}
& v \vDash_{PSL} X!\varphi \\
& \Leftrightarrow v^{1..} \vDash_{PSL} \varphi \text{ (according to definition 18)} \\
& \Leftrightarrow v^{1..} \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\varphi) \text{ (by hypothesis)} \\
& \Leftrightarrow v \vDash_{\mu TL} \bigcirc tr_{PSL \rightarrow \mu TL}(\varphi) \text{ (according to definition 24)} \\
& \Leftrightarrow v \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(X!\varphi) \text{ (according to definition 45)}
\end{aligned}$$

5. $\psi = (\varphi_1 U \varphi_2)$:

$$\begin{aligned}
& \Leftrightarrow v \vDash_{PSL} \varphi U \psi \\
& \Leftrightarrow v \vDash_{\mu TL} \mu X.tr_{PSL \rightarrow \mu TL}(\psi) \vee (tr_{PSL \rightarrow \mu TL}(\varphi \wedge OX)) \text{ (according to lemma 13 and by hypothesis)} \\
& \Leftrightarrow v \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\varphi_1 U \varphi_2) \text{ (according to definition 45)}
\end{aligned}$$

6. $\psi = (r \diamond \rightarrow \varphi)$:

$$\begin{aligned}
& \Leftrightarrow \exists i \text{ s.t. } v^{0..i} \models_{PSL} r \text{ and } v^{i..} \vDash_{PSL} \varphi \text{ (according to definition 18)} \\
& \Leftrightarrow \exists i \text{ s.t. } v^{0..i} \models_{PSL} r \text{ and } v^{i..} \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\varphi) \text{ (by hypothesis)} \\
& \Leftrightarrow \exists i \text{ s.t. } v^{0..i} \in L(tr_{REG \rightarrow NFA}(r)) \text{ and } v^{i..} \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(\varphi) \text{ (according to lemma 9)} \\
& \Leftrightarrow v \vDash_{\mu TL} tr_{NFA \times \mu TL \rightarrow \mu TL}(tr_{REG \rightarrow NFA}(r), \varphi) \text{ (according to lemma 10)} \\
& \Leftrightarrow v \vDash_{\mu TL} tr_{PSL \rightarrow \mu TL}(r \diamond \rightarrow \varphi) \text{ (according to definition 45)}
\end{aligned}$$

7. $\psi = r$:

$$v \models_{PSL} r$$

$$\Leftrightarrow \forall \text{ finite } u \leq v : u \top^\omega \models_{PSL} r \diamond \rightarrow \text{true}$$

$$\Leftrightarrow (\exists \text{ a finite prefix } u \text{ s.t. } u \models_{PSL} r) \text{ or } (\forall \text{ finite prefixes } u \text{ there is a } w \text{ s.t. } uw \models_{PSL} r)$$

(according to lemma 15)

$$\Leftrightarrow (\exists \text{ a finite prefix } u \text{ s.t. } u \in L(\text{tr}_{REG \rightarrow NFA}(r))) \text{ or } (\forall \text{ finite prefixes } u \text{ there is a } w \text{ s.t. } uw \in L(\text{tr}_{REG \rightarrow NFA}(r)))$$

(according to lemma 9)

$$\Leftrightarrow v \in L(\text{tr}_{NFA \rightarrow NBA}(\text{tr}_{REG \rightarrow NFA}(r)))$$

(according to lemma 11)

$$\Leftrightarrow v \models_{\mu TL} \text{tr}_{NBA \rightarrow \mu TL}(\text{tr}_{NFA \rightarrow NBA}(\text{tr}_{REG \rightarrow NFA}(r)))$$

(according to lemma 12)

$$\Leftrightarrow v \models_{\mu TL} \text{tr}_{PSL \rightarrow \mu TL}(r)$$

(according to definition 45)

q.e.d.

3.3 Complexity Analysis

First, we are going to estimate the translation of boolean and regular expressions into NFAs:

- The translation of boolean expressions into NFAs is done in linear runtime, conditional on the length of the boolean expression: every boolean expression is translated into two states with a transition that contains the boolean expression as the guard.

Now, we can analyse the complexity of operations over automata:

- The translations of the concatenation, union and Kleene star operators are linear in the size of the regular expression. The reason for this is that we need to copy - in the case of concatenation and union - new states into the NFA to be built, eventually change the acceptance behaviour and introduce a new initial state with a new transition.

In the worst case, the operations can be exponential in the number of subexpressions.

Consider the following example:

$$\alpha_0 := a \wedge b$$

$$\alpha_{n+1} := \alpha_n \cup \alpha_n$$

α_n has $O(n)$ subexpressions but its syntactical length is $O(2^n)$. An approach to complexity reduction in similar cases will be explained in chapter 5.

- The translation of regular expressions that contain intersection operators is more complex. If there are two NFAs A_1 and A_2 of the size n_1 and n_2 , the intersection NFA has the size $n_1 * n_2$ and can be build in time $O(n_1 * n_2)$. If A_3 is an NFA of size n_3 and we

build the intersection NFA $A_1 \cap A_2 \cap A_3$, this leads to $O(n_1 * n_2 * n_3)$ and so on.

Recapitulatory, we observe that the intersection over k automata of a maximal size n can be executed in $O(n^k)$ which means that the translation of a regular expression requires approximately one exponent per intersection operator. As the translation contains no limitation of intersection operators, we state a complexity of $O(n^n)$.

Translating NFAs into equation systems can be done in linear time: we construct an equation for every transition of the NFA.

Moving on to PSL formula level, we can also observe that the translation of all operators can be done in linear time, by copying equations and priorities into new equation systems and eventually adding new initial equations.

Altogether, if n is the length of a PSL formula φ , it will contain n intersection operators at the most. This leads to a roughly estimated complexity of

$$O(n^n).$$

In section 4.3.2, some auxiliary functions reducing the size of NFAs are explained. Also, the topic will be discussed in chapter 5.

As explained in [Var88] and [Kai97], a μ TL formula of length n can be translated into a Büchi automaton of the size $2^{O(n^2 \log n)}$. This leads to the following theorem:

Theorem 3 (*Complexity of the translation*) *Each PSL formula containing a maximum of k intersection operators can be translated into a μ TL formula of the length $O(n^k)$ and into a Büchi automaton of the size $2^{O(n^{2k} * k * \log n)}$.*

Proof. Follows from this section and [Var88], [Kai97].

4 Implementation

Below the reader will find a description of the implementation of the PSL to μ TL translation that has been developed during this work. After showing data types and functions that have been used, this chapter will finish with examples of formulas that we have tested.

4.1 Framework

The translation of PSL formulas to equation systems was implemented with Objective CAML (Categorically Abstract Maschine Language), a functional programming language that also supports object oriented concepts. Nevertheless, mainly imperative concepts are used the implementation described below.

Objective CAML belongs to the Meta Language family [OCa].

The technical device used for representations of automata is *dotty*, which is part of the Graph Visualisation (Graphviz) package [Dot].

4.2 Data Types

In this section, the data types used for representing PSL formulas, NFAs and equation systems will be explained.

4.2.1 PSL

As PSL is defined over boolean expressions, regular expressions and formulas, we represent each one of those with its own data type. They are a straightforward implementation of the definitions presented in section 2.3, for example

```

type boolExpr = Top
              | Bottom
              | BVar of string
              | BOr of boolExpr list
              | BAnd of boolExpr list
              | BNot of boolExpr

```

Other than explained in definition 1, the conjunction and disjunction over boolean expressions use lists as paramters. In this way, it is easy to model associativity.

Similarly, we define the data types for regular expressions and formulas:

```

type pRegExpr = RBoolExpr of boolExpr
              | Concat of pRegExpr * pRegExpr
              | Union of pRegExpr list
              | Intersection of pRegExpr list
              | Star of pRegExpr

```

In order to keep formulas short and easy to handle, the data type representing PSL formulas also contains the conjunctions of formulas:

```

type pslForm = BoolExpr of boolExpr
              | Or of pslForm list
              | And of pslForm list
              | Not of pslForm
              | StrongNext of pslForm
              | RegNext of pRegExpr * pslForm
              | Until of pslForm * pslForm
              | RegExpr of pRegExpr

```

4.2.2 Automata

In the implementation, automata consist of a record of three components: an array that contains transitions, an array which declares which state is final and which state is not, and an integer indicating the initial state.

All states are numbered so that the states of an automaton can be presented with integers. The array of transitions has an element for every state of the automaton. A transition is implemented as a list and represents a disjunction over pairs of a guard and the state that can be

reached if the guard is fulfilled. Guards can be (negated) atomic propositions or conjunctions or disjunctions over guards.

The *final array* contains a boolean for each state; it is set as *true*, if the corresponding state is final and *false*, if it is not.

```

type guard = Symb of string
           | NSymb of string
           | Disj of (guard list)
           | Conj of (guard list)

type transition = (guard * int) list

type nfa = { delta : transition array;
            final : bool array;
            start : int }

```

Let A be the automaton in figure 3.1. This automaton would be represented by the following as arrays:

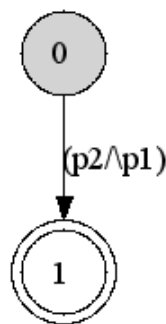
```

transitions : 0 [ (Conj [ Symb p1; Symb p2 ], 1) ]
              1 [ ]
final       : 0 [ false ]
              1 [ true ]

```

As the initial state is q_0 , the NFA shown in figure 3.1 would be represented by the record $\{transitions, final, 0\}$.

For a better readability, NFAs will be displayed graphically:



The initial state is distinguished from other states because it is filled with grey. As usual, final states are highlighted by a double circle and guards are indicated next to the transition arrow.

4.2.3 Equation Systems

As explained in section 2.6.2, equation systems as introduced in section 2.5 and μ TL formulas as introduced in section 2.4 are equi-expressive. The translation of μ TL into equation systems is uniform and equation systems are neither more nor less than a compact representation of μ TL formulas (see section 2.5). Thus, the implementation is based on them.

Equation systems are represented by a record of an initial equation and two arrays, one containing equations and one containing the priority of every corresponding variable. In this way, every variable is represented by an integer.

The data type `equation` represents the definition of terms as explained in section 2.5. Equations can be a (negated) atomic proposition, a state, or a conjunction or disjunction over equations. The priority array contains the priority of each equation.

```

type equation = State of int
                | Symb of string
                | NSymb of string
                | Conj of (equation list)
                | Disj of (equation list)
                | Step of equation

type eqsystem = { prios : int array;
                  eqs : equation array;
                  start : int }

```

In section 2.5.2, there are two examples of equation systems. EQS_4 of this section would be represented by the record $\{equations, priorities, start\}$ such that:

```

equations : 0 [ Symb a ]
            1 [ Symb b ]
            2 [ Disj [b ; (Conj [ a  $\diamond$  2 ] ) ] ]
            3 [ Disj [b ; (Conj [ a  $\diamond$  2 ] ) ] ]
priorities 0 [ 0 ]
            1 [ 0 ]
            2 [ 1 ]
            3 [ 0 ]

```

```
start :      3
```

Henceforth, equation systems will be presented as shown below. The number of an equation is the first sign on the left hand side, followed by the priority of that equation in square brackets and finally, on the right hand side, the equation.

```
Start : 3
0 =[0]= a
1 =[0]= b
2 =[1]= (b \\/ (a /\ <>2))
3 =[0]= <>(b \\/ (a /\ <>2))
```

4.3 Functions

The translation of regular expressions as a part of PSL formulas to equation systems cannot be done directly, we have to make a detour via automata in order to realize the translation explained in chapter 3. Therefore, this chapter will first present functions that are needed to translate boolean and regular expressions into equation systems and then finish with the translation of PSL formulas into equation systems.

4.3.1 Translating Boolean and Regular Expressions to NFAs

Every boolean expression b is translated into an NFA with two states by translating it to a guard and building an NFA with this guard.

The implementation contains a function called `toEquationSystem_reg` which accepts regular expressions as input parameters. If the input parameter is a boolean expression, we have to cast it to a regular expression by using the `RBoolExpr` operator. First, we translate the boolean expression into a guard:

```
let rec transform b = match b with
    BTrue      → Conj []
  | BFalse    → Disj []
  | BOr l     → Disj (List.map transform l)
  | BAnd l    → Conj (List.map transform l)
```

```

| BVar(f)      → Symb(f)
| BNot(f)     → ( match f with ...

```

The negation of guards is applied recursively, depending of the type of the guard. The resulting guard is used to build an NFA, called `atomic_nfa` because it is the elementar entity regarding NFAs. It consists of an initial and an accepting state and a transition from the initial to the accepting state, which is exactly the guard:

```

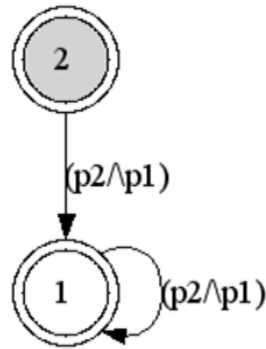
let atomic_NFA g =
  let delta = Array.make 2 [] in
  let final = Array.make 2 false in
  delta.(0) ← [ (g,1) ];
  final.(1) ← true;
  { delta = delta; final = final; start = 0 }

```

Based on these automata, the implementation realizes every operation explained in section 2.2.2:

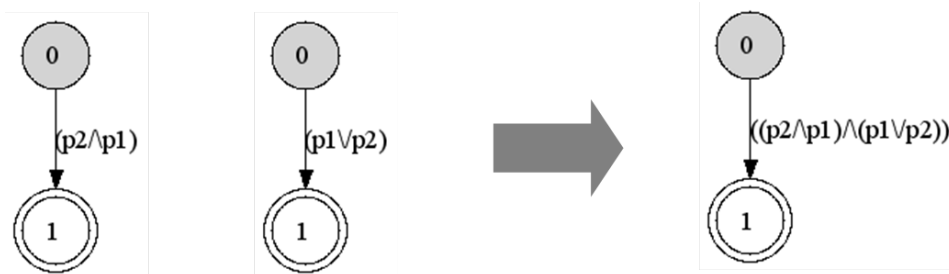
- `val concatenation_nfa : nfa → nfa → nfa`
returns the concatenation NFA A^i of two input NFAs A_1 and A_2 , so that $L(A^i) = L(A_1) \cup L(A_2)$. This implements definition 8.
- `val union_nfa : nfa → nfa → nfa`
takes two input NFAs A_1 and A_2 and delivers an NFA A^U such that $L(A^U) = L(A_1 \cup A_2)$. The construction of the union NFA has been explained in definition 9.
- `val intersection_nfa : nfa → nfa → nfa`
Just like explained in definition 10, this function returns the intersection NFA A^\cap which is constructed over two input automata A_1 and A_2 and returns an NFA that accepts the language $L(A_1) \cap L(A_2)$.
- `val kleene_star_nfa : nfa → nfa`
returns an NFA A^* representing the Kleene star NFA build from an input NFA A_1 such that $L(A^*) = L(A_1)^*$. The implementation corresponds to definition 11.

For example, let A be the NFA explained in section 4.2.2. Applying the `kleene_star_nfa` function leads to the following NFA:



4.3.2 Auxiliary Functions Reducing the Size of NFAs

When operations over automata with similar guards are realized, guards can become very long and have redundant or even contradictory content. For example, if NFA A_1 has a guard of the type $p_1 \wedge p_2$ and NFA A_2 , a guard $p_1 \vee p_2$ and we would apply the intersection operator on them, the following would happen:



The intersection NFA contains a transition $((p_1 \wedge p_2) \wedge (p_1 \vee p_2))$ which of course could be simplified to $p_1 \wedge p_2$. The more guards there are in an automaton and the more operations we apply, the more complicated are the guards in the resulting NFA and of course also the equation systems we translate them into.

Moreover, if we build the intersection NFA over A_1 and an NFA A_3 , containing one transition, namely $\neg p_1 \vee \neg p_2$, it would be translated into an automaton that accepts words $w \in \Sigma$ such that $w^0 \models (p_1 \wedge p_2) \wedge (\neg p_1 \vee \neg p_2)$. This is obviously never the case so that it would be possible to remove the transition and thus reduce the size of the NFA.

To avoid these long or even unsatisfiable guards, the implementation contains a function which transforms guards into disjunctive or conjunctive normal form and removes duplicates. In some cases, the reduction recognizes an unsatisfiable guard and replaces it with *false*. In all other cases, we suppose that the formula is satisfiable and keep the reduced transition. The

algorithm does not necessarily recognize every unsatisfiable guard or deliver a minimal expression. As these problems are co-NP-hard, the aim was not to decide over a minimal or unsatisfiable formula but to implement a function that realizes an approximation to the minimal NFAs.

The functions are named `minimise_DNF` and `minimise_CNF` and require a boolean expression as input parameter. Both methods are part of the μ Sabre project and are applied in the examples in section 4.4 but were they were not part of this work's problem definition. Therefore, the issue is not addressed in detail.

Another approach to keep operations over NFAs small is to eliminate states that can either not reach a final state or cannot be reached from the initial state. The algorithms `eliminate_dead_ends` and `eliminate_needless_states` contain a search through the transition array of the input NFA and replace the transitions of each of those states with \perp . This does evidently not reduce the number of states in an NFA but the operations over guards are shortened. The topic will be discussed in detail in chapter 5.

4.3.3 Translating NFAs to Equation Systems

The translation of NFAs to equation systems combines to steps:

- As the acceptance of regular expressions on the PSL formula level is defined as

$$v \vDash_{PSL} r \Leftrightarrow \forall \text{ finite } u \leq v, u \top^\omega \vDash_{PSL} r \diamond \rightarrow true$$

(see definition 18), we need to adopt the acceptance behaviour of the automata mentioned in section 4.3.1 by adding loops to the final states and afterwards declaring every state final. The idea of this transformation is explained in definitions 43 and 45 and justified in theorem 2.

- Finally, we need a way to transform the NBA into an equation system.

We start by replacing the transitions of every state that cannot reach a final state or that cannot be reached through the initial with *false*, which is logically equivalent to eliminating these states. Thereby, we try to keep the automata as small as possible in order to be more efficient. These applications, as well as a minimization as explained in section 4.3.2, are put together in a function called `simplify_nfa` which takes an NFA as input parameter and applies the

reductions succesively.

Guards as well as equations can be (negated) atomic propositions or disjunctions or conjunctions over guards or equations, respectively. Therefore, guards can directly be translated into the corresponding equation:

```
let rec transform g = match g with
  | Symb f    → Eqsystem.Symb f
  | NSymb f   → Eqsystem.NSymb f
  | Conj xs   → Eqsystem.Conj (List.map transform xs)
  | Disj xs   → Eqsystem.Disj (List.map transform xs)
```

Afterwards, we translate every transition of the form

```
[(guard_1, state_1) ; (guard_2, state_2) ; ... ; (guard_n, state_n)]
```

into an equation

```
Disj [(guard_1, Step(state_1)) ; (guard_2, Step(state_2)) ; ... ;
      (guard_n, Step(state_n))].
```

Finally, we want to interpret the NFA as an NBA and add a transition to every final state, see above. This is realized by appending

```
... ; [ Step ( state_i ) ]
```

to every equation i that emerged from translating the transitions of a state which was final in the NFA.

As there are no fixpoint operators in these equations, all priorities are 0. The initial equation is the equation obtained from translating the transition of the initial state of the NFA. With this, the translation is complete. It is activated via the call `ps1_nfa2eqs` which has an NFA as input parameter and delivers the equivalent equation system.

To give an example, the NFA A^* mentioned in section 4.3.1 would be translated into the following equation system:

```
Start : 2
0 =[0]= (ff /\ <>0)
1 =[0]= (<>1 \\/ ((p2 /\ p1) /\ <>1))
2 =[0]= (<>2 \\/ ((p2 /\ p1) /\ <>1))
```

Referring to chapter 3, this equation system represents the translation $tr_{NBA \rightarrow \mu TL}(tr_{NFA \rightarrow NBA}(A^*))$.

4.3.4 Translating PSL to Equation Systems

As we have shown how to translate boolean and regular expressions into equation systems, we can implement the operators on PSL formula level over equation systems. We divide these functions into three main blocks:

1. negation of equation systems
2. junctions over equation systems (conjunction, disjunction, until)
3. "next" operators

The reader will find a description of each case below.

Negation

The negation of equation systems is realized by negation of every equation. We call the function that reads in an equation system and returns the negated equation system

```
val eqSystemNeg : eqSystem → eqSystem
```

The negation of equation is realized with the subfunction `negateEquation`:

```
let rec negateEquation = function
  | Symb p          → NSymb p
  | NSymb p        → Symb p
  | Disj l         → Conj (List.map negateEquation l)
  | Conj l         → Disj (List.map negateEquation l)
  | Step s         → Step (negateEquation s)
  | State i        → State i
```

We want to build up an equation system in which the model checking game over some word w would be won by \exists if \forall won before negating it and vice versa. So by changing \vee to \wedge and \wedge to \vee , we turn around the situations in which \exists and \forall would choose the next configuration. Finally, we increase the priority of every negated equation by one which makes even priorities odd and odd priorities even. Thus, if we are in an infinite play, we change the type of the outermost variable with respect to $<_{\varphi}$ which causes again that the opposite player will win. The initial equation stays the same.

Example: This is the equation system explained in section 4.2.3, negated:

```

Start : 3
0 =[1]= -a
1 =[1]= -b
2 =[2]= (-b /\ (-a \/ <>2))
3 =[1]= <>(-b /\ (-a \/ <>2))

```

Junctions

The conjunction, disjunction and until operators are all implemented with respect to the same principle which is depicted in figure 4.1. If EQS_1 and EQS_2 are two equation systems, these functions are accomplished by creating a new equation system and copying EQS_1 and EQS_2 into it and adding a new initial equation e_0 . Let $e_{0,1}$ and $e_{0,2}$ be the initial equations of EQS_1 and EQS_2 , then e_0 would be

$e_{0,1} \vee e_{0,2}$, in the case of a disjunction

$e_{0,1} \wedge e_{0,2}$ in the case of a conjunction, or

$e_{0,2} \vee (e_{0,1} \wedge \diamond e_0)$ if we want to realize the until operator.

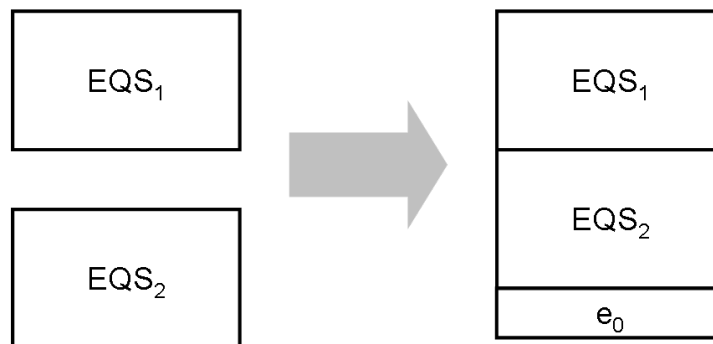


Figure 4.1: Junction of equation systems

In order to compute the copying of an equation system, we need a function that increases the size of the second equation system by the size of the first equation system. Thereby we avoid wrong references inside the new equation system. The implementation contains the following auxiliary function:

```

let rec increaseAllStateNames i = function
  State j          → State (i+j)
| Disj fs          → Disj (List.map (increaseAllStateNames i) fs)
| Conj fs          → Conj (List.map (increaseAllStateNames i) fs)
| f                → f

```

Afterwards, we initialise two new arrays which will represent the equations and priorities of the new equation system. We copy the priorities and the equations belonging to the input equation systems successively, the second with application of `increaseAllStateNames`, to the new arrays.

The function realizing the junctions is called `eqSystemJunction f p eqs1 eqs2` where `eqs1` and `eqs2` are the input parameters, `f` is the respective function and `p` determines the priority of the new initial equation. Originating from `eqSystemJunction`, the following functions are part of the implementation:

- `val eqSystemOr : eqSystem → eqSystem → eqSystem`
realising the disjunction over two equation Systems,
- `val eqSystemAnd : eqSystem → eqSystem → eqSystem`
realising the conjunction over two equation Systems and
- `val eqSystemUntil : eqSystem → eqSystem → eqSystem`
realising the U operator over two equation Systems.

For example, the equation system below realizes the disjunction over the equation systems explained in sections 4.2.3 and 4.3.4:

```

0 =[0]= a
1 =[0]= b
2 =[1]= (b ∨ (a ∧ <>2))
3 =[0]= <>(b ∨ (a ∧ <>2))
4 =[1]= -a
5 =[1]= -b
6 =[2]= (-b ∧ (-a ∨ <>2))
7 =[1]= <>(-b ∧ (-a ∨ <>2))
8 =[0]= (<>(b ∨ (a ∧ <>2)) ∨ <>(-b ∧ (-a ∨ <>2)))

```

The equations 0 to 3 are a copy of the equation system mentioned in section 4.2.3, representing the formula $X!(aUb)$ and the equations 4 to 7 are a copy of the equation system from section 4.3.4, which is the negated equation system build from $X!(aUb)$. The initial equation 8 represents $X!(aUb) \vee X!\neg(aUb)$.

Next Operators

There are two different next operators in the PSL syntax, the next operator for formulas after regular expressions ($\diamond\rightarrow$) and after formulas ($X!$). The implementation of the $X!$ operator is quite simple: we define a new initial equation which adds a " \diamond " to the old initial equation. The function is called

- `val eqSystemNext : eqSystem → eqSystem`

and has an equation system as input parameter and returns the equation system with a new initial equation.

For example, if we want to apply the $X!$ operator to the equation system representing $(a \vee b)$, namely:

```
Start : 4
0 =[0]= a
1 =[0]= b
2 =[0]= ff
3 =[0]= (b \\/ ff)
4 =[0]= (a \\/ (b \\/ ff))
```

we get:

```
Start : 5
0 =[0]= a
1 =[0]= b
2 =[0]= ff
3 =[0]= (b \\/ ff)
4 =[0]= (a \\/ (b \\/ ff))
5 =[0]= <>(a \\/ (b \\/ ff))
```

The implementation of the $\diamond\rightarrow$ operator requires some intermediate steps. Let $EQS_\varphi = (E_\varphi, \Phi_\varphi, e_{0,\varphi})$ be the equation system obtained from translating φ as explained in this section and let $EQS_A = (E_A, \Phi_A, e_{0,A})$ be the equation system which represents the NFA obtained from translating r into an NFA A (see section 4.3.1), and A into an equation system (see section 4.3.3).

First, we modify every $e_A \in E_A$ by applying the following function:

$$\zeta(e_A) = \begin{cases} Step(a) \wedge Step(b) & , \text{ if } e_A = Step(a \wedge b) \\ Step(a) \vee Step(b) & , \text{ if } e_A = Step(a \vee b) \\ e_A & \text{ otherwise.} \end{cases}$$

Afterwards, every subterm $Step(i)$ such that i is final in A is replaced by $Disj[Step(i) ; e_{0,\varphi}]$ in every equation $\zeta(e_A), e_A \in E_A$. Thus, for every time that A could reach an accepting state, the equation system contains a subterm leading to the translation of φ , namely EQS_φ .

Now, we copy all equations $\zeta(e_A), e_A \in E_a$ and all equations $e_\varphi \in E_\varphi$ into the new equation system EQS_{\diamondrightarrow} .

Finally, if 0 is final state of A , we add $Disj[e_{0,\varphi}; e_{0,A}]$ to $\zeta(e_{0,A})$ in order to assure that if A accepts ϵ , EQS_{\diamondrightarrow} realizes φ .

The function described above is realized in the implementation by the function

```
val eqSystemRegNext : eqSystem → eqSystem → eqSystem
```

and is equivalent to $tr_{PSL \rightarrow \mu TL}(r \diamondrightarrow \varphi)$, see definition 45.

4.4 Empirical Results

In order to describe and analyse the characteristics of the implementation, three examples will be presented in this section. We focused on the translation of regular expressions because all other formulas can be translated in linear time.

The first example is a bitwise counter for arbitrary many bits, the second a function computing leap years, and the third the language words that contain non-overlapping repetitions of substrings. All examples are implemented with the help of regular expressions and will be illustrated with NFAs and equation systems resulting from translating these NFAs.

The test results presented in this section were observed on a PC with 512 MB RAM and a 3 GHz Pentium 4 processor.

4.4.1 N Bit Counter

When we say "n bit counter", we want to compute an automaton that accepts words of the form

$$\begin{array}{l|l} b_0 & 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ \dots \\ b_1 & 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ \dots \\ b_2 & 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ \dots \\ & \vdots \\ b_n & \dots \end{array}$$

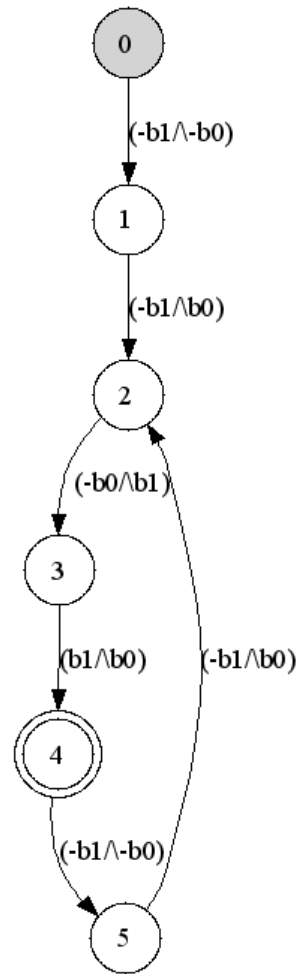


Figure 4.2: NFA representing a 2 bit counter

so that, for example, the 4 bit counter produces all binary representations of the numbers 0 to $2^4 - 1 = 15$, etc. The function is named `counter`, has an integer (`n`) as input parameter and will produce the regular expression that represents the `n` bit counter.

If b_i stands for the i th bit, the following formulas are helper functions we need to compute the counter.

First, we can define b_0 , which will be the same in every `n` bit counter:

- $bit_0 := (-b_0; b_0)^*$

Now, we can define the i th bit for $i > 0$, which always depends on the $i - 1$ th bit: we want the i th bit to change from 0 to 1 and 1 to 0, after the $i - 1$ th bit has changed from 0 to 1 and is going to change back to 0. This can be realized by defining

bits	states	time (min:sec)
1	4	<0:01
2	6	<0:01
3	10	<0:01
4	18	<0:01
5	34	<0:01
6	66	0:01
7	130	0:01
8	258	0:03
9	514	0:08
10	1.026	0:21
11	2.050	0:55
12	4.098	2:17
13	8.194	5:29
14	16.386	13:16
15	32.770	30:59
16	65.538	74:02
17	131.074	176:25

Table 4.1: Size of NFAs representing n bit counters

- $aux_{i-1} := (\neg b_{i-1}; (\neg b_{i-1})^*); (b_{i-1}; (b_{i-1})^*)$
- $bit_i := (((\neg b_i; (\neg b_i)^*) \cap (aux_{i-1}); ((b_i; (b_i)^*) \cap (aux_{i-1})^*$

Finally, we define a regular expression setting all bits to 0 at the start:

- $start := \bigwedge \neg b_i$

From these formulas, we can build up the regular expression we are looking for:

$$counter := (start) \cap \left(\bigcap_{i=0}^n (bit_i) \right)$$

The NFA corresponding to a 2 bit counter is displayed in figure 4.2.

Table 4.1 shows a list of different sized counters. All automata have exactly $2^n + 2$ states. As the reader can see, the NFAs for 1 to 10 bit counters were built in less than a minute, counters with 16 or more bits require even more than an hour. These results are consistent with

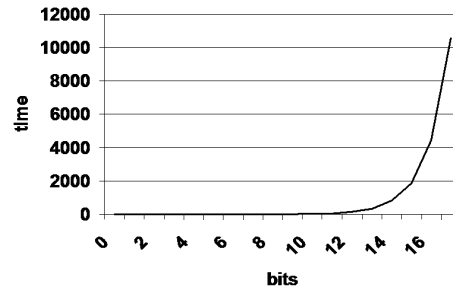


Figure 4.3: Time used to calculate n bit counters

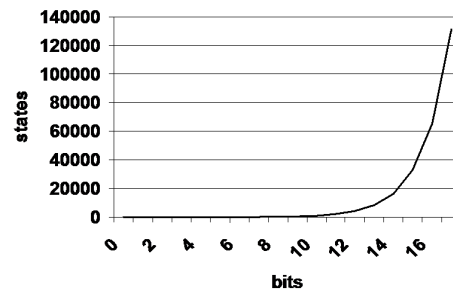


Figure 4.4: Number of states of n bit counters

the complexity analysis in section 3.3: as counters with more bits contain more intersection operators, the time needed to translate them grows exponentially.

The test series was stopped after the 17 bit counter because of memory problems. Figures 4.3 and 4.4 visualize the results.

4.4.2 Leap Years

The second example analysed is a regular expression describing the characteristics of leap years. A year is a leap year, if the whole-numbered remainder is zero when dividing it by four, although not zero when dividing it by 100 but again zero when dividing it by 400. For example, 2000 and 2004 are leap years, but 2100 is not.

The implementation contains a function called `leap_year` which has no input parameters. It builds up a regular expression r over the alphabet $\{0, \dots, 9\}$ such that

$$L(r) = \{w \mid w \text{ represents a decimal leap year when reading it from right to left}\}.$$

Again, we need a few auxiliary functions to prepare. First, we need to specify that we want to analyse the sequence of numbers that the leap year consists of one after the other. Thus, we implement a formula which ensures that no formulas like - for example - $3 \wedge 5$ would be accepted:

- $one_by_one := (\bigvee_{i=0}^9 (i) \wedge \bigwedge_{j=0}^9 (j \mid_{i \neq j}))^*$

Defining formulas to distinguish odd and even numbers, such as

- $odd := 1 \vee 3 \vee 5 \vee 7 \vee 9$

- $even := 0 \vee 2 \vee 4 \vee 6 \vee 8$

It is now easy to define formulas to specify the remainders when dividing a year by 4:

- $0_mod_4 := (0 \vee 4 \vee 8) \cup ((true; true^*); ((even ; (0 \vee 4 \vee 8)) \vee (odd ; (2 \vee 6))))$

where *true* is some expression which will always be fulfilled, such as

$$true := 0 \vee \neg 0$$

Similarly, we define 0_mod_400 and $not_0_mod_100$. The regular expression defining leap years can now be build over these auxiliaries:

$$leap_year := (one_by_one ; one_by_one^*) \cap (0_mod_400 \cup (0_mod_4 \cap not_0_mod_100))$$

The resulting NFA consists of 353 states and is therefore not displayed here. The translation took 2:18 minutes.

This example shows that the translation as implemented can lead to automata and equation systems that are a lot bigger than the optimal solution. Consider figure 4.5 which shows an NFA that accepts decimal representations of leap years and only consists of 9 states.

q_0 is the initial state and q_5 as well as q_8 are accepting states. Runs of the form $q_0q_1q_2q_3q_5 \dots$ or $q_0q_1q_2q_4q_5 \dots$ accept words that represent numbers that are divisible by 400, such as 0080 or 0021, etc. If a run passes from q_0 to q_8 via q_6 or q_7 , words that do not start with zero but are divisible by four are accepted, for example 4201 or 6322. Finally, the automaton accepts words that are divisible by four and start with zero with runs of the form $q_0q_1q_8 \dots$

The optimization of the implemented algorithms will be discussed in chapter 5.

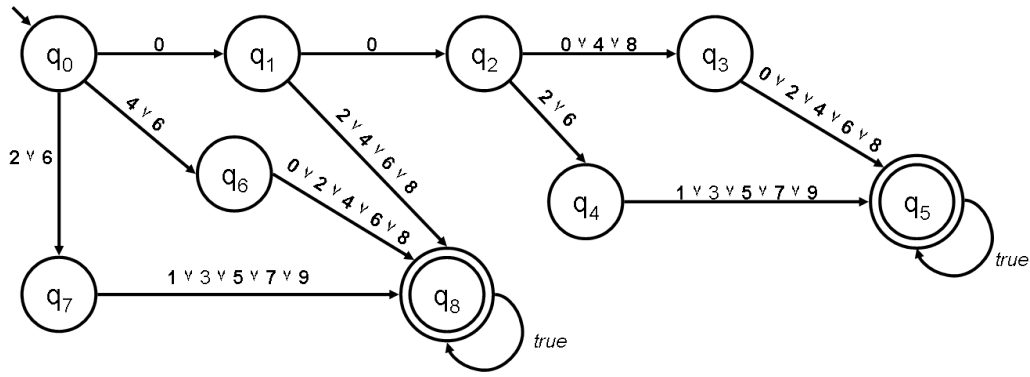


Figure 4.5: Automaton accepting decimal representations of leap years

4.4.3 Non-overlapping Repetition of Substrings in a String

The last example that will be presented in this work deals with the problem of finding substrings that occur repeatedly without overlapping each other in a string. For example, the word 01101 contains the substring 01 repeatedly and without overlap.

The aim is to compute an automaton A^n such that

$$L(A^n) = \{w \mid w \text{ has 2 substrings } w^{i..j} \text{ and } w^{k..l} \text{ of length } n \text{ s.t. } 0 \leq i < j < k < l \leq n \text{ and } w^{i..j} = w^{k..l}\}.$$

As explained in [KZ02] and [Pet02], the language we are looking for can be modelled with the aid of regular expressions, either with or without using the intersection operator. Both approaches and a concluding comparison will be presented below.

First, we consider the regular expression r_1^n such that $L(r_1^n) = L(A^n)$ and r_1^n contains *no* intersection operators. In the implementation, the corresponding function is named `overlap_without_intersection`, has an integer n as input parameter and uses the alphabet $\Sigma = \{0, 1\}$. If W^n is the set of all words of length n , it produces the regular expression

$$r_1^n = \bigcup_{w^n \in W^n} (0 \vee 1)^* ; w^n ; (0 \vee 1)^* ; w^n ; (0 \vee 1)^*$$

so that $L(r_1^n) = L(A^n)$.

Secondly, we present a regular expression r_2^n such that $L(r_2^n) = L(A^n)$ and r_2^n consists of operators including the intersection. The function is called `overlap_with_intersection` and uses the same input parameter and alphabet as `overlap_without_intersection`.

We use the notation $\{0, 1\}^{\min k}$ to describe $\{0, 1\}^l$ such that $k \leq l$. Furthermore, let

$$\text{disjunct } i \ x := \{0, 1\}^i ; x ; \{0, 1\}^{\min n-1} ; x ; \{0, 1\}^{n-i-1}$$

and

$$\text{intersection} := \{(\text{disjunct } i \ 0) \cup (\text{disjunct } i \ 1) \mid i \leq n\}$$

Then

$$r_2^n := (0 \vee 1)^* ; \text{intersection} ; (0 \vee 1)^*$$

and $L(r_2^n) = L(A^n)$.

Table 4.2 shows the results observed when building automata over the regular expressions r_1^n (without intersection) and r_2^n (including the intersection operator) for non-overlapping substrings of the length $n = 1, 2, \dots, 7$.

All automata, whether including intersection operations or not, are exponential in n , so that they are a significant example to analyse the performance of the translation.

n	r_1^n		r_2^n	
	states	time (min:sec)	states	time (min:sec)
1	35	<0:01	29	<0:01
2	87	<0:01	45	<0:01
3	207	<0:01	245	<0:01
4	479	<0:01	1.805	0:01
5	1.087	<0:01	16.629	0:03
6	2.431	0:01	181.597	0:35
7	5.375	0:01	2.271.589	10:53

Table 4.2: Size of NFAs representing the language of words that contain non-overlapping substrings of length n

As mentioned in section 4.3.2, the implementation contains functions that reduce the number of operations over transitions by replacing certain transitions with \perp . This leads to a number of needless or unproductive states, which were counted and subtracted from the total number of states in order to receive the number of productive state for $n = 1$ and $n = 2$.

For $n = 1$, the automaton built for r_1^n contains 11 productive states, versus 9 states in the NFA built for r_2^n . For $n = 2$, the respective sizes are 29 and 31 states. The number of *productive* states does not vary remarkably although the number of states produced and thus the time used

grows a lot slower in relation to n when the intersection operator is used.

These results re-emphasize the fact that time and space complexity of the translation as presented in this thesis could be considerably enhanced by improving the minimization of automata, such as deleting non-productive states.

5 Conclusion

This thesis provides a method to translate the PSL core language LTL_{WR} into the linear time μ -calculus in exponential time and space complexity in the length of the input formula.

The PSL specification contains a combination of regular expressions and formulas and therefore the translation of PSL to μ TL requires constructions over automata as auxiliary means. In the absence of the intersection operator in the input formula, the translation can be realized in polynomial complexity.

5.1 Results Achieved

The algorithms used to translate PSL into μ TL are explained and their correctness is proved in chapter 3. Formulas produced by this translation are always alternation-free and thus relatively easy to handle.

As explained in theorem 3, by translating PSL into μ TL and building a Büchi automaton over the translated formula, we showed that a PSL formula of length n with k intersection operators can be translated into a Büchi automaton in $2^{O(n^{2k} * k * \log(n))}$. Compared to a complexity of $2^{2^{O(n)}}$ as stated in [SBD05], we observe a slight improvement.

Relating to the μ -Sabre project, the aim of adding a plain and easily understandable input language to the existend bounded model checker was achieved. Further work on bounded model checking for the alternation-free μ -calculus and Büchi automata can be found in [HJK⁺06] and [SBD05].

5.2 Outlook

Major improvement of the complexity of the translation as implemented could be reached by further elaboration of the automata minimization.

On the one hand, memory requirements can be reduced by extending the existing function

replacing the transitions of all unnecessary states \perp (see section 4.3.2) with a function that deletes these states and thus reduces the size of the constructed automata.

On the other hand, the translation as implemented is composed by translating all subexpressions of the input formula which can obviously lead to repetitive and thereby redundant parts of automata or equation systems. This topic has been pointed at in section 3.3. Memorizing the subexpressions that have already been translated and reusing them in future calculation can lower the complexity.

A more general approach to improve the efficiency of the implementation would be to analyse different ways of interpreting regular expressions without using automata constructions. For example, [Brz64] introduces *derivatives* of regular expressions which follow the idea of interpreting regular expressions as trees. It might be of interest to define a data structure for such derivatives and compare the resulting possibilities of analysing regular expressions to the translation explained in this work.

A Bibliography

- [AO] Inc. Accellera Organization. *Property Specification Language : Reference Manual*.
- [BB89] B. Banieqbal and H. Barringer. Temporal logic with fixed points. In *Proc. Coll. on Temporal Logic in Specification*, volume 398 of *LNCS*, pages 62–73. Springer, 1989.
- [BKP86] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *Conf. Record of the 13th Annual ACM Symp. on Principles of Programming Languages, POPL'86*, pages 173–183. ACM, ACM, 1986.
- [Brz64] J.A. Brzozowski. Derivates of Regular Expression. *Journal of the ACM*, 11:481–494, 1964.
- [Büc62] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Congress on Logic, Method, and Philosophy of Science*, pages 1–12, Stanford, CA, USA, 1962. Stanford University Press.
- [CBRZ01] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
- [CE82] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons from branching time temporal logic. *Lecture Notes Comp. Sci.*, 131:52–71, 1982.
- [Cor] Intel Corporation. FDIV Replacement Program. <http://support.intel.com/support/processors/pentium/fdiv/>.
- [Dax06] C. Dax. Games for the linear time μ -calculus. Master's thesis, Dep. of Computer Science, University of Munich, 2006. available from http://www.tcs.ifi.lmu.de/lehre/da_fopra/Christian_Dax.pdf.
- [Dot] Graphviz, Doty. <http://www.graphviz.org>.

- [EJ91] Emerson and Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.
- [Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 996–1072. Elsevier and MIT Press, New York, USA, 1990.
- [FTD] Financial Times Deutschland.
http://www.ftd.de/technik/medien_internet/108916.html.
- [GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. The temporal analysis of fairness. In *Proc. 7th Symp. on Principles of Programming Languages, POPL'80*, pages 163–173. ACM, 1980.
- [HJK⁺06] K. Heljanko, T. Junttila, M. Keinänen, M. Lange, and T. Latvala. Bounded model checking for weak alternating büchi automata. In *Proc. 18th Int. Conf. on Computer Aided Verification, CAV'06*, volume 4144 of *LNCS*, pages 95–108. Springer, 2006.
- [HU80] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1980.
- [JJ83] J.P. Queille and J. Sifakis. Fairness and related properties in transition systems. *Acta Informatica*, 19:195–220, 1983.
- [Kai97] R. Kaivola. *Using Automata to Characterise Fixed Point Temporal Logics*. PhD thesis, LFCS, Division of Informatics, The University of Edinburgh, 1997. Tech. Rep. ECS-LFCS-97-356.
- [Kam68] H. W. Kamp. *On tense logic and the theory of order*. PhD thesis, Univ. of California, 1968.
- [Kön36] D. König. *Theorie der endlichen und unendlichen Graphen. Kombinatorische Topologie der Streckenkomplexe*, volume 16 of *Mathematik und ihre Anwendungen in Monographien und Lehrbüchern*. Leipzig: Akademische Verlagsgesellschaft, 1936.
- [Koz83] D. Kozen. Results on the propositional μ -calculus. *TCS*, 27:333–354, December 1983.

- [KZ02] Kupferman and Zuhovitzky. An improved algorithm for the membership problem for extended regular expressions. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 2002.
- [MuS] Mu-Sabre. <http://www.tcs.ifi.lmu.de/musabre/>.
- [OCa] Objective Caml. <http://caml.inria.fr/ocaml/>.
- [Pet02] Petersen. The membership problem for regular expressions with intersection is complete in LOGCFL. In *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, 2002.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symp. on Foundations of Computer Science, FOCS'77*, pages 46–57, Providence, RI, USA, 1977. IEEE.
- [SBD05] D. Fisman A.Griesmayer I. Pill S. Ruah S. Ben-David, R.Bloem. Automata Construction Algorithms Optimized for PSL (deliverable 3.2/4), 2005.
- [Sti95] C. Stirling. Local model checking games. In *Proc. 6th Conf. on Concurrency Theory, CONCUR'95*, volume 962 of *LNCS*, pages 1–11. Springer, 1995.
- [Sti01] C. Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer, 2001.
- [Tho79] W. Thomas. Star-free regular sets of ω -sequences. *Information and Control*, 42(2):148–156, 1979.
- [Tho97] W. Thomas. Languages, automata and logic. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer, 1997.
- [Var88] M. Y. Vardi. A temporal fixpoint calculus. In ACM, editor, *Proc. Conf. on Principles of Programming Languages, POPL'88*, pages 250–259, NY, USA, 1988. ACM Press.
- [Var96] M. Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.