Máster en Investigación en Informática
Facultad de Informática
Universidad Complutense de Madrid

Two Algorithms for Model Checking of Regular Linear Temporal Logic

Julian Samborski-Forlese

Director: Miguel Palomino Tarjuelo
Colaborador externo: César Sánchez

2010-2011

Calificación obtenida: 8

Autorización de difusión

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, el presente Trabajo Fin de Máster: *Two Algorithms for Model Checking of Regular Linear Temporal Logic*, realizado durante el curso académico 2010-2011 bajo la dirección de Miguel Palomino Tarjuelo y con la colaboración externa de dirección de César Sánchez en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

En Madrid, a los 9 días del mes de febrero de 2012,

Julian Samborski-Forlese
X4523081Q

Miguel Palomino Tarjuelo
Director

A mi mamá Analía,
porque así de bella es la vida.

Resumen

Este trabajo estudia el problema de *ver si una fórmula es satisfacible*—the satisfiability problem—y el problema de *model checking* para Regular Linear Temporal Logic (RLTL).

En el contexto de la *verificación de programas*, el estudio de técnicas de verificación de *sistemas reactivos*—aquellos que interactúan con el entorno y responden a estímulos—ha ganado especial interés dentro de la comunidad científica. El *model checking* es una técnica automática muy poderosa para verificar la corrección de estos sistemas.

Un model checker está típicamente compuesto por un *lenguaje de descripción* para modelar sistemas, un *lenguaje de especificación* para codificar propiedades y un *método de verificación*. Para verificar que un sistema satisface una propiedad, el usuario describe el modelo, afirma las hipótesis y el proceso de verificación descubre si dichas hipótesis son validas en el modelo. Una de las características más importantes del model checking es la capacidad de producir contraejemplos cuando una propiedad no se satisface en el sistema.

El model checking está basado en *lógica temporal*, en la cuál el *valor de verdad* de una fórmula depende de la *evolución* del sistema. Linear Temporal Logic (LTL) es una lógica temporal que en la actualidad es ampliamente aceptada como formalismo para la especificación y verificación de sistemas concurrentes y sistemas reactivos. No todos los lenguajes ω-regulares pueden ser expresados por LTL y esta falta de expresividad suele aparecer en la práctica. Al mismo tiempo, algunos estudios señalan que las expresiones regulares, en combinación con LTL, son muy convenientes para escribir especificaciones formales.

Regular Linear Temporal Logic (RLTL), ha sido propuesta como una extensión de LTL con constructores basados en expresiones regulares . RLTL extiende la expresividad de LTL a todos los lenguajes ω-regulares, tiene una base formal sólida y complejidad óptima para extensiones de LTL.

En este trabajo presentamos dos nuevos enfoques al problema de ver si una formula es satisfacible (satisfiability problem) y al problema del model checking para RLTL. Por un lado, este trabajo extiende a Regular Linear Temporal Logic las ideas de *bounded model checking*—una técnica de verificación de propiedades temporales lineales basada en procedimientos SAT—desarrollada para Linear Temporal Logic. Como parte de este esfuerzo se presenta una traducción de expresiones RLTL a fórmulas SAT. Por el otro lado, este trabajo estudia el enfoque al model checking para RLTL basado en la *teoría de autómatas*—el cuál reduce este problema de verificación a la construcción de autómatas y problemas de decisión sobre autómatas. Se presenta una traducción de expresiones RLTL a *strong alternating parity automata*, junto con una traducción de los autómatas resultantes a *non-deterministic Büchi automata*. Además, introducimos la noción de *stratified automata*, la cuál nos permite realizar una traducción basada en ránkings mucho más eficiente. Los resultados experimentales obtenidos son muy alentadores.

Palabras clave: *verificación de programas, model checking, bounded model checking, regular linear temporal logic, teoría de autómatas, complejidad*

Abstract

This thesis studies the satisfiability and model checking problem for Regular Linear Temporal Logic (RLTL).

In the context of *program verification*, model checking is an automatic, powerful technique to verify the correctness of systems. In particular, the verification of *reactive systems* has gathered much research effort. Reactive systems are systems that interacts with their environments and respond to stimuli. A model checker is typically comprised by a *description language* for modeling systems, a *specification language* for encoding properties and a *verification method*. To verify that a system verifies a property, the user describes a model, asserts hypotheses, and the verification process discovers whether such hypotheses are valid on the model. One of the most important features of model checking is that counterexamples can be produced when a property fails to be satisfied in the system.

Model checking is based on *temporal logic*, where the *truth* depends on the *evolution* of the system. Linear Temporal Logic (LTL) is a temporal logic that is now a widely accepted formalism for the specification and verification of concurrent and reactive systems. LTL cannot express all ω-regular properties and this lack of expressivity seems to surface in practice. At the same time, some studies point out that regular expressions are very convenient in addition to LTL in formal specifications.

Regular Linear Temporal Logic (RLTL), a logic for the temporal frame, was proposed as an extension of LTL with constructs based on regular expressions. RLTL extends the expressive power of LTL to all ω-regular languages and has a formal foundation with well-studied complexity results.

In this work, we provide two novel approaches to the satisfiability and model checking problems for RLTL. On the one hand, this work extends the ideas of *bounded model checking*—a verification technique for linear time properties based on propositional decision procedures (SAT)—developed for standard Linear Temporal Logic into Regular Linear Temporal Logic. As part of this effort, a translation from RLTL expressions into SAT formulas is presented. On the other hand, this work studies the *automata-theoretic* approach to model checking for RLTL—which reduces this verification problem to automata constructions and automata decision problems. A translation from RLTL expressions into *strong alternating parity automata* is presented along with a translation from the resulting automata into *non-deterministic Büchi automata*. We also introduce the notion of stratified automata, leading to a more efficient ranking translation with encouraging experimental results.

Keywords: *program verification, model checking, bounded model checking, regular linear temporal logic, automata theory, complexity*

Agradecimientos

En primer lugar quisiera expresar mi gratitud infinita a César Sánchez, mi guía en este maravilloso mundo que es la investigación. César no sólo es un gran investigador al que admiro profundamente, sino también una excelente persona con la cuál disfruto cada momento compartido. Su constante ayuda y consejos han hecho este trabajo posible.

Gracias a Miguel Palomino por aceptar ser mi director de máster. Sus consejos y sugerencias han mejorado generosamente la presentación de este trabajo.

A Gilles Barthe y César Kunz por darme la oportunidad y enseñarme a dar los primeros pasos en esta profesión. A ambos, un sentido agradecimiento.

Me gustaría agradecer también al Instituto IMDEA Software, y a todo su personal, por brindarme el excepcional entorno y las herramientas necesarias para realizar mi trabajo diario. Un especial gracias a Tania y Paola, su simpatía y esmero hacen nuestras vidas más fáciles.

A mi viejo y mi hermano, mi pequeña gran familia, por confiar en mí y hacerme el aguante, por ser fuertes ante la adversidad y demostrarme que la vida continúa a pesar de todo, porque son Grandes y los amo con el alma, porque merecen esto y mucho más.

A mi madrina Andrea, mi segunda mamá, por estar siempre, por quererme tanto. Mis papás hicieron una gran elección.

A mis hoy compañeros de ruta, Alejandro, Carolina y Javier, por el apoyo continuo, por brindarme su amistad y por todas esas fantásticas experiencias que compartimos viviendo juntos.

A mis amigos de siempre, Mauro, Dante y Gustavo, por estar cuando los necesité, por compartir inolvidables momentos, por enseñarme tanto. Son increíbles.

A Juanma, por ser un gran amigo y bancarme en tantas. A César K., por ser una gran persona. Quizás el tipo con mayor sentido del humor que conozca. Madrid no hubiese sido tan divertido sin ellos.

Gracias a todos los que de alguna u otra manera han estado y no encontraron su nombre entre estas líneas. No ha sido adrede. Todos saben lo mucho que los aprecio y lo agradecido que les estoy.

Un eterno gracias a mi mamá, esa bella mujer que me enseñó que la magia existe, porque a su lado aprendí lo mágica que es la vida. Fue ella quien me enseñó a perseguir mis sueños y a nunca bajar los brazos ante la adversidad. Su incansable lucha ha sido, es y será el mayor ejemplo de fortaleza que jamás conoceré. Alegre, dulce y divertida, su impronta quedará para siempre en nuestros corazones. Me diste todo mamá, te debo más aún. Nunca mis palabras serán suficientes para expresarte mi amor.

Julián

Contents

1

Introduction

This work studies the satisfiability and model checking problems for Regular Linear Temporal Logic and in particular the development of new algorithms to efficiently model-check expressions in this logic. This chapter includes an overview of the contributions of this thesis along with some discussion on the state of the art.

1.1 Overview

Reactive Systems are systems that maintain an ongoing interaction with their environment. From this point of view, reactive systems are everywhere. Ranging from small cooking artifacts and watches to complex software systems, such as robots, operating systems, aircrafts, industrial plants and the like. They all share the common principle of being systems that respond or react to external stimuli, whether normal user-generated ones—such as pressing a button or prompting a command—or environmental-generated ones—such as temperature raising or daylight disappearing—or abnormal ones—such as a power failure. Reactive systems can be also required to respond in real-time or behave synchronously with other components. In many cases, those systems play an extremely important role in our society, as is the case with *safety-critical systems*, and their correctness must be verified.

Program Verification There is a great advantage in being able to verify the correctness of reactive systems. Since the early works of Robert Floyd and Tony Hoare in the late 1960's, the field of program verification has gathered much research effort. The ultimate goal of this discipline is to devise proofs for the correctness of programs. However, it has become quickly apparent that finding such proofs is extremely hard, usually much more so than writing the program itself. Moreover, program proofs are often very long and tedious, which makes them very difficult to compute by hand. For those reasons, a lot of research has been put into the development of automatic methods for software verification. The idea is to use computer programs to find and / or validate proofs of correctness of other programs.

Formal verification techniques can be seen as having three main components:

- a *framework for modeling systems*, typically a description language,
- a *specification language* for describing the properties to be verified,
- a *verification method* to establish the correspondence between the description and the specification of a system, i.e., whether the description satisfies the specification.

In general, verification techniques can be *proof-based* or *model-based*. In a proof-based approach, the description of the system is a set of formulas Γ and the specification is another formula φ. The verification

is done by trying to find a proof that $\Gamma \vdash \varphi$. In the model-based approach, the system is represented as a model $\mathcal{M}$ and the specification is again represented by a formula φ. The verification is performed by computing whether the model satisfies the formula ($\mathcal{M} \models \varphi$).

Model Checking In this thesis we focus on a model-based, automatic property-verification approach called *model checking*. This technique is intended to be used for *concurrent, reactive* systems and originated as a post-development methodology. In this setting, the user describes a model, asserts hypotheses, and the verification process discovers whether such hypotheses are valid on the model. If it is not the case, counterexamples consisting of execution traces can be produced.

Model checking is based on *temporal logic*. The idea of temporal logic is that a formula is not *statically* true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others. The static notion of truth is replaced by a *dynamic* one. In this dynamic notion the evolution of the system from state to state may change the truth values of formulas. The verification process consist of the three following steps:

- describe the model $\mathcal{M}$ using the description language of the model checker.
- write the temporal logic formula φ using the specification language of the model checker.
- run the model checker with inputs $\mathcal{M}$ and φ.

The model checker answers *yes* or *no* depending on the satisfaction of the formula. As we said, if the answer is no, the model checker provides a traces of the failing behavior. In the design and debugging of systems, automatic generation of *counter traces* is a tool that plays a very important role. Since model checking is a model-based technique, we are not concerned with semantic entailment ($\Gamma \models \varphi$), or with proof theory ($\Gamma \vdash \varphi$), in this thesis we work solely with the notion of satisfaction, i.e. the satisfaction relation between a model and a formula ($\mathcal{M} \models \varphi$).

Linear Temporal Logic and its Limitations In his seminal paper in FOCS'77, Pnueli proposed Linear Temporal Logic (LTL) [Pnu77, MP95] as a specification language for reactive systems. LTL is a temporal logic, with connectives allowing references to the future. It models time as an infinite sequence of states. This sequence is sometimes called a computation path, or simply a path.

Wolper showed that LTL cannot express all ω-regular properties. Even though LTL is now a widely accepted formalism for the specification and verification of concurrent and reactive systems, the lack of expressivity seems to surface in practice. To alleviate the expressivity problem, Wolper suggested *extended temporal logic* (a tradition followed by Vardi, Kupferman and others) in which new operators are defined using automata, and instantiated using alphabet substitution. The main drawbacks of this logic are that (1) in order to obtain the full expressivity, an infinite number of operators is needed; and (2) operator composition is implemented using alphabet substitution, which is cumbersome for specification engineers. An alternative approach to the expressivity problem is to adapt the modal μ-calculus to the linear frame. In this approach one needs to use fix-point binders to describe temporal properties, which again tends to make typical specifications cumbersome. At the same time, some studies point out that regular expressions are very convenient in addition to LTL in formal specifications, partly because practitioners are familiar with regular expressions, partly because specifications are more natural. Some approaches to extend the μ-calculus with regular expressions, or dynamic logics like RPL, increase the expressivity beyond ω-regular languages, at the price of undecidability.

The popularity of regular expressions led also to their inclusion in the industry standard specification language PSL. While decision procedures and their complexities for full PSL are still an area of active research, we know that the fragment of PSL that contains LTL and semi-extended regular expressions leads to EXPSPACE satisfiability and model checking problems.

Regular Linear Temporal Logic In 2007, Martin Leuker and César Sánchez [LS07, SL10] proposed Regular Linear Temporal Logic (RLTL), a logic for the temporal frame, as an extension of LTL with

constructs based on regular expressions. RLTL extends the expressive power of LTL to all ω-regular languages.

RLTL has a simple syntax with a small signature, and generalizes LTL and regular expressions while keeping the complexity under control. It can be also extended with past [SL10].

RLTL can offer practitioners a simple specification logic that extends LTL and regular expressions to all regular languages, has a formal foundation with well-studied complexity results. More precisely, the complexity of the satisfiability and model checking problems for RLTL is PSPACE-complete, optimal for extension of LTL [SC85].

The field of computer-aided verification is faced with a fundamental undecidability problem. In general, checking whether a computer program satisfies its specification is undecidable. Computer-aided verification techniques circumvent this undecidability barrier by exploiting sound abstractions which over-approximate the behavior of the original program. The idea is to substitute the concrete program with an abstract, more simpler version, in such a way that any property that holds on the abstraction also holds on the concrete program. Note that this technique is sound but not complete in general, so there is no contradiction with the general undecidability of program verification.

In this thesis we develop new algorithms for the decidability and model checking problems for RLTL. To that end, we use two different approaches. The first one is based on symbolic model checking and propositional decision procedures (SAT), and the second one is based on automata theory.

Symbolic model checking [BCM+92] uses a Boolean encoding of the finite state machine and states. It has been reported to be able to verify systems with large number of states (more than 10^{20}). Binary Decision Trees (BDDs for short), a canonical form for Boolean expressions, have traditionally been used as the underlying representation for symbolic model checkers. In general, model checkers based on BDDs performs very well. For large systems, however, the BDDs generated during model checking becomes too large for currently available computers. Propositional decision procedures also operates on Boolean expressions but do not use canonical forms. Thus, they do not suffer from the potential state explosion of BDDs, being able to handle SAT problems with thousand of variables.

The *automata-theoretic approach* to model checking reduces this verification problem to automata constructions and automata decision problems. The verification process begins by translating the negation of the formula into an equivalent automaton on infinite words. This automaton accepts all the traces that violate the specification. Then, the automaton is composed with the system description using synchronous product composition, for which a non-emptiness check answers whether the system admits some counterexample trace.

Bounded model checking BMC was introduced by Biere et al. [BCCZ99]. It is a symbolic model checking technique based on SAT procedures. Its basic idea consist on considering counterexamples of a particular length k and produce a propositional Boolean formula that is satisfiable if and only if such a counterexample exists. For more details you can also read the work of Latvala et al. [LBHJ04] titled "Simple bounded LTL model checking". In [BHJ+06] they show linear encodings of LTL into Boolean formulas for bounded model checking. Bounded model checking for LTL has become a very active research topic since its beginnings and lots of impressive results have been published. We now briefly enumerate the most important of them. The work reported in [AS04] study which properties one can verify with bounded model checking. [AS06] studies termination criteria for bounded model checking. Benedetti and Cimatti [BC03] studies BMC for past LTL (for past LTL see [LBHJ05, HJL05]). A survey of BMC is presented in [BCC+03]. An approach to bounded model checking for weak alternating automata appears in [HJK+06], see also [BCP+06]. Bounded model checking for all regular properties is studied by Jehle et al. in [JJLR06]. Results about the complexity of BMC can be found in [CKOS04]. Separated normal forms can be applied to accelerate BMC [JS07] as suggested also in [FSW02]. Other references are the works from Strichman [Str04] and Sebastiani et al. [STV05].

1.2 Contribtutions

The purpose of this thesis is to find new efficient algorithms for the satisfiability and model checking problems for RLTL. Our aim is to develop algorithms that uses distinct verification approaches to analyse which best suits RLTL. We summarize here the main contributions of this work.

Our first contribution is the development of an algorithm to translate RLTL expressions into SAT formulas to analyse the behavior of ultimately periodic words (infinite words expressible as a finite prefix followed by a finite postfix repeated infinitely many times). The motivation for this work was to investigate how to apply the concepts of *Bounded model checking* to RLTL. This work has a number of interesting characteristics. First, we define a bounded semantics that is sound w.r.t. the original unbounded semantics of RLTL. Second, in an effort to improve the bounded model checking for RLTL and develop more efficient algorithms, we define a semantics based on three-valued logic which is more precise, and potentially more powerful, than the bi-valued semantics. This opens the door to develop an algorithm capable of discover failures on the system more quickly. It even improves the original BMC algorithm for the LTL part of RLTL. Third, we have proved some nice properties of derivatives of basic regular expressions (regular expressions that do not accept the empty language). Finally, we introduce a translation from RLTL expressions into propositional formulas for ultimately periodic words. In contrast to the original BMC, instead of defining the semantics and the translation based on paths of a system, we define them on infinite words. It is worth to note that both approaches are equivalent and each one can be rewritten in terms of the other.

Our second contribution is a novel translation from the logic RLTL into alternating parity automata using only colors 0, 1 and 2, based on a bottom-up construction of specular pairs accepting complement languages. This work was inspired on previous works on the topic and the necessity to implement an efficient translation as part of a more ambitious goal that aims to extend antichains algorithms to solve the emptiness problem for RLTL. Inspired by the duality in the translation we introduce universal sequential operators that enrich the logic with negation normal forms. We also show that the resulting automata enjoy some nice properties. Then, we studie translations of the resulting automata into *non-deterministic Büchi* (NBW). We introduce the concept of stratified automata and obtain a more efficient ranking translation that preserves the alphabet between alternating automata and NBW. A prototype has been developed, and thus we present some experimental results that are very encouraging.

1.3 Structure of the Thesis

This work is structured as follows:

- In *Chapter 2* we recall some of the basic concepts and notations about numbers, sets, graphs, logic, automata, an so on. In Section 2.6 we define RLTL, the underlying logic of all this work. This chapter fixes the formal notation used throughout this thesis.

- In *Chapter 3* we introduce Bounded model checking for RLTL. We present here the new definitions of the bounded semantics, recall some concepts on derivatives of regular expressions along with the introduction of some nice properties of derivatives of basic regular expressions, and the SAT-based translation of RLTL expressions.

- In *Chapter 4* we present the translation from RLTL into strong parity automata (APW) and into non-deterministic Büchi (NBW). We introduce the notion of stratified automata and provide a better ranking translation. We also provide some experimental results to show the effectiveness of our method.

- In *Chapter 5* we present the conclusions and future work.

2

Preliminaries

This chapter reviews the basic notation used throughout the rest of the thesis.

2.1 Sets, Functions, and Relations

We use $\mathbb{N}$ to denote the set of *natural numbers* $\{0, 1, 2, \dots\}$, and $\mathbb{N}_1$ to denote the set of *positive natural numbers* $\{1, 2, 3, \dots\}$. We denote by $\mathbb{Z}$ the set of *integer numbers* $\mathbb{N} \cup \{-1, -2, \dots\}$. With $\varnothing$ we denote the empty set. For any set S we denote by 2^S the *power-set* of S, i.e., the set of subsets of S. We denote by $|S| \in \mathbb{N} \cup \{+\infty\}$ the *cardinality* of a set S. For any subset $X \subseteq S$ of a set S, whenever S is clear from the context, we denote by $\overline{X} \stackrel{\text{def}}{=} S \setminus X$ the *complement* of X in S. A *partition* of a set S is a set of nonempty subsets of S such that every element $s \in S$ is in exactly one of these subsets, i.e., $P \subseteq 2^S$ is a partition of S iff $P = \{S_1, \dots, S_n\}$ s.t. $\bigcup P = S$, $S_i \cap S_j = \varnothing$ for all $i \neq j$, and $S_i \neq \varnothing$. In what follows, we may sometimes make use of Church's *lambda notation* to anonymously define functions. For instance, $\lambda x \cdot 2x$ is such an anonymous "lambda-style" definition. For a function $f : A \to B$, we call A the *domain* of f, and B the *codomain* of f, which are denoted $\mathsf{dom}(f)$ and $\mathsf{codom}(f)$, respectively. The *image* of a function $f : A \to B$ is the set $\mathsf{img}(f) \stackrel{\text{def}}{=} \{f(a) \mid a \in A\}$.

2.1.1 Binary Relations

A *binary relation* over a set S is a set of pairs $R \subseteq S \times S$. A binary relation is *reflexive* if an only if for each $s \in S$ we have that $\langle s, s \rangle \in R$; it is *symmetric* if and only if for each pair $\langle s_1, s_2 \rangle \in R$ we have that $\langle s_2, s_1 \rangle \in R$; it is *antisymmetric* if and only if for each pair $\langle s_1, s_2 \rangle \in R$ such that $\langle s_2, s_1 \rangle \in R$ it is also the case that $s_1 = s_2$; it is total if and only for each pair $s_1 \in S$, $s_2 \in S$ we either have that $\langle s_1, s_2 \rangle \in R$ or $\langle s_2, s_1 \rangle \in R$; finally, it is transitive if and only if for each triple s_1, s_2, s_3 such that $\langle s_1, s_2 \rangle \in R$ and $\langle s_2, s_3 \rangle \in R$, it is also the case that $\langle s_1, s_3 \rangle \in R$.

Binary relations are often denoted using non-alphabetic symbols and the infix notation. Let $\sim \subseteq S \times S$; the expression $s_1 \sim s_2$ is equivalent to $\langle s_1, s_2 \rangle \in \sim$, and $s_1 \nsim s_2$ is equivalent to $\langle s_1, s_2 \rangle \notin \sim$. To make the notations more intuitive, we always use symmetric symbols for symmetric relations and vice-versa. Moreover, the infix notation allows to use symbol relations backwards, e.g., the expression $s_1 \preceq s_2$ is equivalent to $s_2 \succeq s_1$ for any binary relation $\preceq$.

Let $R \subseteq S \times S$ be a binary relation; we denote by R_0 the set of pairs $\{\langle s, s \rangle \mid s \in S\}$, and for every $i \in \mathbb{N}_1$ we denote by R_1 the relation:

$$\{\langle s_1, s_3 \rangle \in S \times S \mid \exists s_2 \in S : \langle s_1, s_2 \rangle \in R_{i-1} \wedge \langle s_2, s3 \rangle \in R\}$$

The *transitive closure* of R, denoted R^+, is the relation:

$$\left\{\langle s_1, s_2\rangle \in S \times S \mid \exists i \in \mathbb{N}_1 : \langle s_1, s_2\rangle \in R^i\right\}$$

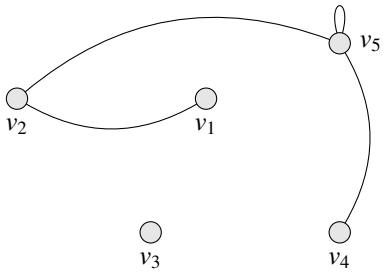
The *reflexive and transitive closure* of R, denoted R^*, is the relation $R^0 \cup R^+$. A *preorder* is a reflexive and transitive binary relation. A preorder that is also symmetric is an *equivalence*. A preorder that is also antisymmetric is a *partial order*.

2.2 Graphs

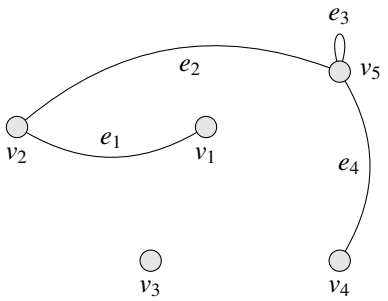
In this section, we briefly remember some of the basic concepts regarding graph theory that will be used in the remaining of this thesis.

2.2.1 Undirected Graphs

A *graph* $G = (V, E)$ is a pair of sets where V is a set of *nodes* and E is a set of *edges*, formed by unordered pairs of nodes. For example, if $V = \{v_1, \ldots, v_5\}$ and $E = \{(v_1, v_2), (v_2, v_5), (v_5, v_5), (v_5, v_4)\}$, the graph $G = (V, E)$ is represented as



We often label the edges with letters (e.g., $a, b, c \ldots$ or $e_1, e_2, \ldots$). In the example, if we label the edges as follows:



then $E = (e_1, e_2, e_3, e_4)$. The vertices u and v are end vertices of the edge (u, v). An edge of the form (v, v) is called a *loop*. A graph with no edges is *empty*, a graph with no vertices is a *null graph*, and graph with only one vertex is *trivial*. We say that two edges are *adjacent* if they share a common vertex. If two vertices are connected by an edge, we say that they are *adjacent*. The *degree* of a vertex is the number of edges that has such a vertex as an end node. A vertex with no incident edges (i.e., with degree 0) is an *isolated vertex*. In our example, e_3 is a loop, e_2 and e_4 are adjacent, v_1 and v_2 are adjacent, the degree of v_2 is 2 and v_3 is an isolated vertex.

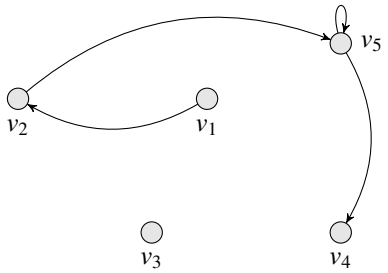
Subgraph A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if G' is a graph, $V' \subset V$ and $E' \subset E$.

Walks and Paths A *walk* in a graph $G = (V, E)$ is sequence of vertices $v_0, \ldots, v_n$ so that $v_i \in V$ and $(v_i, v_{i+1}) \in E$ for every $i \in 0 \ldots n - 1$. If $v_0, \ldots, v_n$ are distinct, we call this a *path*.

Connected A graph $G = (V, E)$ is connected if for every $u, v \in V$ there is a walk from u to v. A *component* of G is a maximal nonempty connected subgraph of G.

2.2.2 Directed Graphs

A *directed graph* or *digraph* is a graph $D = (V, E)$ where E is a set of ordered pairs. If $u, v \in V$, there is an edge directed from u to v if $(u, v) \in E$. We also say that u and v are the *tail* and the *head* of the edge, respectively. Both are end nodes of (u, v). As an example of a digraph, let $D = (V, E)$ where V and E are the sets from the previous example with the only difference that now E is interpreted as a set of *ordered pairs*. Graph D can be drawn as follows.



The *degree* of a vertex v is divided into the *outdegree* and the *indegree*. The oudegree is the number of edges with tail v, and the indegree is the number of edges with head v.

Subgraph This concept is analogous the one for undirected graphs.

Directed Walks and Paths A *directed walk* in a digraph $D = (V, E)$ is a sequence $v_0, \ldots, v_n$ so that $v_i \in V$ and for every $0 \leq i < n$, $(v_i, v_{i+1}) \in E$. A *directed path* is a walk where $v_0, \ldots, v_n$ are distinct.

Strongly Connected A digraph $D = (V, E)$ is *strongly connected* if for every $u, v \in V$ there is a directed walk from u to v. A *strong component* of a digraph D is a maximal strongly connected subgraph of D.

Acyclic A directed acyclic graph if *DAG* is a directed graph with no directed cycles.

2.2.3 Infinite Graphs

An *infinite graph* $G = (V, E)$ is a graph where V and E each have infinite cardinality. The concepts described for finite graphs can be extended to the infinite case.

2.3 Boolean Logic

In this section, we quickly recall a number of definitions and notations regarding Boolean logic that are needed in the sequel.

2.3.1 Boolean Functions

We denote the set of *Boolean truth values* as $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$. In general, a set of logical truth values can contain more than two elements (e.g., to accommodate a third, uncertain possibility), and can even be infinite (e.g., the real numbers between 0 and 1); Boolean logic however, deals only with the binary world of *true* and *false*.

Let $\mathcal{X}$ be a finite set of elements called propositions. A *valuation* over $\mathcal{X}$ is a set $v \subseteq \mathcal{X}$ which identifies a *truth assignment* of each proposition in $\mathcal{X}$; by convention, the propositions in v are seen as assigned to

true and the propositions in $\mathcal{X} \setminus v$ are seen as assigned to **false**. For any valuation $v \in 2^{\mathcal{X}}$ and proposition $p \in \mathcal{X}$, we use the notations $v|_{p=\textbf{true}} \stackrel{\text{def}}{=} v \cup \{p\}$ or $v|_{p=\textbf{false}} = v \setminus \{p\}$ interchangeably.

A *Boolean function* is a function f of type $f : 2^{\mathcal{X}} \to \mathbb{B}$ for some finite set of propositions $\mathcal{X}$. We refer to the cardinality $|\mathcal{X}|$ as the *arity* of a Boolean function. For any Boolean function $f : 2^{\mathcal{X}} \to \mathbb{B}$, with $p \in \mathcal{X}$ and $t \in \mathbb{B}$, we define $f|_{p=t}$ to be the function $f' : 2^{\mathcal{X}} \to \mathbb{B}$ such that $f'(v) = f(v|_{p=t})$ for every valuation $v \in 2^{\mathcal{X}}$. We denote the set of Boolean functions over $\mathcal{X}$ by $\mathsf{BF}(\mathcal{X})$.

The *efficient representation and manipulation* of Boolean functions is a central problem in both computer science and engineering. The most explicit, and the simplest representation of a Boolean function is the *truth table*, which contains one row for each of the possible $2^{|\mathcal{X}|}$ inputs. The remainder of this section reviews a more interesting representation of Boolean functions, namely *propositional Boolean formulas*.

2.3.2 Propositional Boolean Formulas

Let us first formally define the syntax of propositional Boolean formulas.

Definition 2.3.1 (Syntax of Propositional Boolean Formulas)**.** Let $\mathcal{X}$ be a finite set for Boolean propositions, and let $p \in \mathcal{X}$. The set of syntactically correct *propositional Boolean formulas* is defined recursively with the following grammar:

$$\varphi ::= \textbf{true} \;\Big|\; \textbf{false} \;\Big|\; p \;\Big|\; \neg\varphi \;\Big|\; \varphi \vee \varphi \;\Big|\; \varphi \wedge \varphi$$

We denote the set of propositional Boolean formulas over $\mathcal{X}$ by $\mathcal{B}(\mathcal{X})$.

The above definition introduces basic *logical connectives*. The symbol $\neg$ is the unary *negation*, $\vee$ is the *disjunction* connective and $\wedge$ is the *conjunction* connective. Additionally, the following *syntactic shorthands* can be defined: $(\varphi \Rightarrow \psi) \stackrel{\text{def}}{=} (\neg\varphi \vee \psi)$ is the *logical implication* connective (it reads "φ implies ψ",) and $(\varphi \Leftrightarrow \phi) \stackrel{\text{def}}{=} (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$ is the *logical equivalence* connective (it reads "φ if and only if ψ".)

As stated previously, propositional Boolean formulas are syntactic representations of Boolean functions. The represented Boolean function is obtained by induction on the structure of the formula, as formalized in the following definition.

Definition 2.3.2 (Semantics of Propositional Boolean Formulas)**.** Let $\mathcal{X}$ be a finite set of Boolean propositions, and φ be a propositional Boolean formula over $\mathcal{X}$. The semantics of the formula φ, denoted $[\![\varphi]\!]$, is the Boolean function $f : 2^{\mathcal{X}} \to \mathbb{B}$ such that, for every valuation $v \in 2^{\mathcal{X}}$:

$$\begin{aligned}
[\![\textbf{true}]\!](v) &= \textbf{true} \\
[\![\textbf{false}]\!](v) &= \textbf{false} \\
[\![p]\!](v) &= \textbf{true} \text{ iff } p \in v \text{ (assuming } p \in \mathcal{X}) \\
[\![\neg\varphi]\!](v) &= \textbf{true} \text{ iff } [\![\varphi]\!](v) = \textbf{false} \\
[\![\varphi_1 \vee \varphi_2]\!](v) &= \textbf{true} \text{ iff } [\![\varphi_1]\!](v) = \textbf{true} \text{ or } [\![\varphi_2]\!](v) = \textbf{true} \\
[\![\varphi_1 \wedge \varphi_2]\!](v) &= \textbf{true} \text{ iff } [\![\varphi_1]\!](v) = \textbf{true} \text{ and } [\![\varphi_2]\!](v) = \textbf{true}
\end{aligned}$$

We see from the above definition that propositional Boolean formulas associate to each valuation of their propositions the value **true** or **false**. When $[\![\varphi]\!](v) = \textbf{true}$ we say that the valuation v *satisfies* the formula φ, which we denote more compactly by $v \vDash \varphi$. Conversely, when $[\![\varphi]\!](v) = \textbf{false}$ then v does not satisfy φ which is simply written $v \nvDash \varphi$. A formula φ is said to be *satisfiable* iff it is satisfied by at least by one valuation, or equivalently iff $[\![\varphi]\!] \neq \lambda v \cdot \textbf{false}$. Dually, a formula φ is *valid* iff it is satisfied by every valuation, or equivalently iff $[\![\varphi]\!] = \lambda v \cdot \textbf{true}$. Deciding whether a given propositional Boolean formula is satisfiable or valid are fundamental NP-Complete and co-NP-Complete problems, respectively [CKS81]. We denote by $\mathsf{Sat}(\varphi) \stackrel{\text{def}}{=} \{v \in 2^{\mathcal{X}} \mid v \vDash \varphi\}$ the set of valuations which satisfy a formula φ.

2.3.3 Positive Boolean Formulas

In the reminder of this thesis, we will often manipulate *positive Boolean formulas*, which are Boolean formulas that do not contain negation symbols. More precisely, given a set of propositions $\mathcal{X}$, these formulas are built from **true**, **false** and elements of $\mathcal{X}$, using $\wedge$ and $\vee$. We use $\mathcal{B}^+(\mathcal{X})$ to denote the set of positive Boolean formulas over $\mathcal{X}$.

A subset $M \in \mathcal{X}$ is a *model* of a positive Boolean formula θ if M satisfies θ. The set of models of θ is denoted by $Mod(\theta)$. We say that a model $M \in Mod(\theta)$ is a *minimal model* of θ, and denote $M \in mod(\theta)$, whenever no strict subset of M is also in $Mod(\theta)$. For example, given the set $Q = \{q_0, q_1, q_2, q_3\}$, the formula $\theta_1 = (q_1 \wedge q_2) \vee q_3$ is a $\mathcal{B}^+(Q)$ formula. The minimal models of θ_1 are $\{q_1, q_2\}$ and $\{q_3\}$. The subset $\{q_1, q_3\} \in Q$ is a model of θ_1 but not a minimal model.

Given a positive Boolean formula θ its *dual formula* $\widetilde{\theta}$ can be obtained by switching $\wedge$ and $\vee$, and switching **true** and **false**. For example, the dual of θ_1 above is $\widetilde{\theta_1} = (q_1 \vee q_2) \wedge q_3$, or equivalently in disjunctive normal form $\widetilde{\theta_1} = (q_1 \wedge q_3) \vee (q_2 \wedge q_3)$. The minimal models of $\widetilde{\theta_1}$ are $\{q_1, q_3\}$ and $\{q_2, q_3\}$.

2.4 Alphabets, Words, Languages and Regular Expressions

In this section we review the most important definitions and introduce some notation related to languages and regular expressions. These concepts are intensively used in the reminder of this thesis.

2.4.1 Languages

An *alphabet* is a nonempty set and is denoted by Σ. We call *letters* to the elements of such set.

Finite Words A finite word is a finite sequence $w = \sigma_0, \ldots, \sigma_n$ of letters taken from Σ. The empty word, i.e., the word containing zero letters, is denoted λ. We denote by Σ^* the set of all finite words over Σ. A finite-word language is a finite or infinite set of finite words, i.e., a set $L \subseteq \Sigma^*$. We also call Σ^* the finite-word universal language that contains all the words of finite length, and we call the empty set of words $\varnothing$ the empty language.

Infinite Words These notions are extended to infinite words in the following natural way. An infinite word is an infinite sequence $w = \sigma_0, \sigma_1, \ldots.$ of letters from Σ. We denote by Σ^ω the set of all infinite words over Σ. An infinite-word language is a set of infinite words, i.e., a set $L \subseteq \Sigma^\omega$. We call Σ^ω the infinite-word universal language.

The length of a word w, denoted $|w|$, is the number of occurrences of symbols in w. The length of an infinite word is simply $+\infty$. The position of a letter in a finite or infinite word is a natural number, the first letter being at position zero. We denote the letter of position i in a finite or infinite word w by w_i. For any word w such that $|w| > i$, we denote by $w[i \ldots]$ the suffix of the word w, starting at and including position i. Note that a suffix has always a strictly positive length. If i and j are such that $i, j \leq |w|$ for a given word w, we define $w_{ij} = w_i \ldots w_{j-1}$. For completeness, if $j < i$, $w_{ij} = \lambda$.

In this thesis we do not consider languages that contain both finite and infinite words.

2.4.2 Regular Languages

The *concatenation* of two finite words is the word formed by juxtaposing the two words together, i.e., writing the fist word immediately followed by the second word, with no space in between. For example, if $\Sigma = a, b$ is an alphabet and $r = abb$ and $s = ab$ are two words over Σ, the concatenation of r and s, denoted by rs is $abbab$. The concatenation of two languages of finite words $L_1, L_2 \subseteq \Sigma^*$, denoted $L_1 L_2$ is the set $L_1 L_2 = \{uv \mid u \in L_1 \wedge v \in L_2\}$.

Let n be a nonnegative integer and w a word over an alphabet Σ. Then w^n is a word over Σ defined by

(*i*) $w^0 = \lambda$, and

(*ii*) $w^n = ww^{n-1}$, for $n > 0$.

For an integer $n > 0$ and a language L, the n$^{\text{th}}$ power of L, denoted L^n, is defined by

(*i*) $L^0 = \{\lambda\}$, and

(*ii*) $L^n = L^{n-1}L$, for n > 0.

The star (Kleene closure) of a language L, denoted L^*, is the set

$$\bigcup_{i=0}^{\infty} L^i$$

Definition 2.4.1 (Regular Language)**.** Let Σ be an alphabet. The set of *regular languages* over Σ is defined recursively as follows.

(*i*) $\varnothing$, $\{\lambda\}$ and $\{p\}$ for any letter $p \in \Sigma$ are regular languages.

(*ii*) If L_1 and L_2 are regular languages, then $L_1 \cup L2$, L_1L_2 and $L_1{}^*$ are regular languages.

(*iii*): Nothing is a regular languages unless it is obtained from the above two clauses.

Any language that belongs to this set is called a regular language over Σ.

For example, if $\Sigma = \{a, b\}$, since $\{a\}$ and $\{b\}$ are regular languages, $\{a, b\}$ $(= \{a\} \cup \{b\})$ and $\{ab\}$ $(= \{a\}\{b\})$ are regular languages. Also, $\{a\}^*$, the set of words consisting of a's, is also a regular language.

2.4.3 Regular Expressions

We use regular expressions to denote regular languages. They can succinctly represent regular languages and operations on them. The formal definition is as follows.

Definition 2.4.2 (Regular Expression)**.** Let Σ be an alphabet. A regular expression is defined recursively as follows:

(*i*) $\varnothing$, λ and a for all letters $a \in \Sigma$ are regular expressions.

(*ii*) If r and s are regular expressions, the so are $r + s$, $r\ ;\ s$ and r^*.

(*iii*) Nothing else is a regular expression unless its being so follows from a finite number of applications of Rules (*i*) and (*ii*).

Correspondence Between Regular Languages and Regular Expressions The set of languages accepted by regular expressions is exactly that of regular languages.

Remark. Note that the Kleene star can be defined as a binary operator instead of a unary operator as follows: $r * s = r^*\ ;\ s$. In the reminder of this thesis we will use the binary version of the Kleene star.

2.5 Finite Automata

In this section, we review the basic formalisms needed to *encode computation systems* and *represent formal languages*.

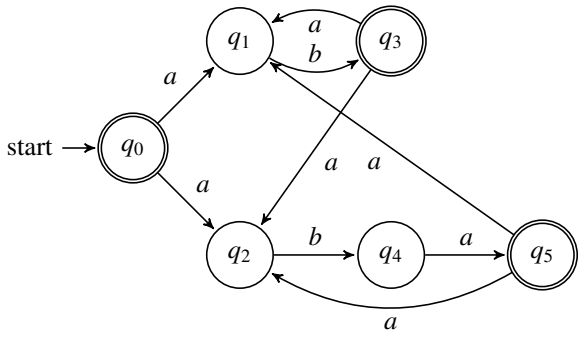


Figure 2.1: A simple FSM.

2.5.1 FSM and Automata

In the representation of formal languages, the *finite state machine* (or FSM) is one of the most fundamental tools. An FSM can be described as a graph whose vertices are called *states*, and whose edges are called *transitions*, and which identifies a set of *initial states* along with a set of *final states* (we simply call *states* to those that are neither initial nor final). In addition, the transitions of an FSM are labeled by symbols taken from a finite alphabet.

Definition 2.5.1 (Finite State Machine)**.** A finite state machine is a tuple $\langle \Sigma, Q, \rightarrow, I, F \rangle$ where:

- $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ is a finite *alphabet*.
- $Q = \{q_1, \ldots, q_n\}$ is a finite set of *states*.
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is a labeled *transition relation*.
- $I \subseteq Q$ is a finite set of *initial states*.
- $F \subseteq Q$ is a finite set of *final states*.

Figure 2.1 shows a simple FSM with alphabet $\{a, b\}$, states $\{q_0, \ldots, q_5\}$, initial state q_0, final state set $F = \{q_0, q_3, q_5\}$ and transition relation $\rightarrow = \{\langle q_0, a, q_1\rangle, \langle q_0, a, q_2\rangle, \langle q_1, b, q_3\rangle, \langle q_2, b, q_4\rangle, \langle q_3, a, q_1\rangle, \langle q_3, a, q_2\rangle, \langle q_4, a, q_5\rangle, \langle q_5, a, q_1\rangle, \langle q_5, a, q_2\rangle\}$.

The transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ of FSM is inherently *non deterministic*, in the sense that there can be several edges labeled with the same alphabet symbol which go from a single state to several distinct states. On the contrary, *deterministic* FSM are such that for any state, the number of outgoing edges labelled with the same symbol is at most one (i.e., $\forall q \in Q, \forall \sigma \in \Sigma : |\{q' \mid \langle q, \sigma, q' \rangle \in \rightarrow\}| \leq 1$. Furthermore, deterministic FSM must have exactly one initial state, i.e., I is a singleton set.

We say that an FSM is *complete* iff $\forall q \in Q, \forall \sigma \in \Sigma : |\{q' \mid \langle q, \sigma q' \rangle \in \rightarrow\}| \geq 1$. In other words, a complete FSM contains no deadlocks. That is, each state has always at least one outgoing transition for every alphabet symbol. Without loss of generality we shall assume in the remainder of this thesis that all FSM that we manipulate are always complete, unless otherwise stated.

Finite state machines admit either finite or infinite executions called *runs*. A run of an FSM is a finite or infinite path inside the machine which follows the transition relation. Runs which originate from the initial state are called initial. Moreover, every run of a finite state machine is classified as either *accepting* or *rejecting* (non-accepting). In the case of finite runs, the acceptance criterion is simple: a finite run is accepting iff it ends in a final state. For infinite runs, the most common is the so-called Büchi acceptance condition—for which a run is accepting for Büchi iff it visits final states *infinitely often*—but many others acceptance conditions have been studied in automata theory, such as Street, Rabin, Parity, generalized Büchi, co-Büchi and others.

Definition 2.5.2 (Run of a Finite State Machine)**.** Let $M = \langle \Sigma, Q, I, \rightarrow, F \rangle$ be an FSM and let $w \in \Sigma^* \cup \Sigma^\omega$ be a word. A *run* of the finite state machine M over w is a sequence of states $\rho \in Q^* \cup Q^\omega$ such that

$|\rho| = |w|$ and for every position $i \in \{0, \ldots, |w| - 1\}$ we have that $\langle \rho[i], w_i, \rho[i+1] \rangle \in \rightarrow$. A run is *initial* iff $\rho[0] \in I$. The set of initial runs of M over w is denoted $init(M, w)$. A *finite* run ρ over a finite word w is *accepting* iff $\rho[|w|] \in F$. For any infinite run ρ, we denote by $inf(\rho) \subseteq Q$ the set of states which appear infinitely often in ρ. An infinite run ρ is *accepting for Büchi* iff $inf(\rho) \cap F \neq \emptyset$.

The completeness assumption for FSM implies that for every word $w \in \Sigma^* \cup \Sigma^\omega$ a deterministic finite state machine admits *exactly one initial run* over w. Deterministic FSM therefore admit two natural language interpretations: one for finite words, and one for infinite words.

In the case of *non-deterministic* FSM, each finite or infinite word corresponds to at least one, but possibly many runs. For non-deterministic FSM, we therefore duplicate each language interpretation in terms of *existential* or *universal* acceptance. Intuitively, a word is accepted in an existential language interpretation if and only if the machine admits at least one initial accepting run on that word; dually, a word w is accepted in the universal language interpretation if and only if every initial run over w is accepting.

A *finite state automaton* is a finite state machine paired with a language interpretation. For instance, a *non-deterministic finite automaton* (NFA) is a non-deterministic finite state machine interpreted existentially over finite words; a *universal Büchi automaton* (UBW) is a non deterministic finite state machine interpreted universally with a Büchi acceptance condition, and so on. We denote by $\mathcal{L}(\mathcal{A})$ the language of the automaton $\mathcal{A}$.

2.5.2 Alternating Automata

In the early 1980's, Chandra et al. [CKS81] introduced the concept of *alternation* in computer science as a generalization of the existencial and universal language interpretations for finite state machines.

Alternating finite automata can be defined by partitioning the set of states into *existencial* and *universal* states. In this setting, an *alternating* language interpretation of the underlying FSM comes out naturally: the quantification over initial accepting runs alternates between existencial ans universal quantifiers depending on the nature of the current state.

A more conveniente, although much less intuitive, way of defining alternating automata is to *augment* the syntax of finite state machines by replacing the transition relation by an *alternating transition function* that maps each state and alphabet symbol to a *positive Boolean function* that maps each state and alphabet symbol to a *positive Boolean function* over the set of states Q. This allows each state to be either fully existential (by using only disjunctions) or fully universal (by using only conjunctions), but it is also possible to freely mix conjunctions and disjunctions, which makes for a convenient and clean syntax.

Definition 2.5.3 (Alternating Automaton)**.** An *alternating automaton* is a tuple $\mathcal{A} : \langle \Sigma, Q, \delta, I, F \rangle$ where:

- $\Sigma = \{\sigma_0, \ldots, \sigma_k\}$ is a finite *alphabet*.
- $Q = \{q_1, \ldots, q_n\}$ is a finite set of *states*.
- $\delta : Q \times \Sigma \rightarrow \mathcal{B}^+(Q)$ is an *alternating transition function*.
- $I \in \mathcal{B}^+(Q)$ is the *initial condition*.
- F is the *acceptance condition*.

Definition 2.5.4 (Alternating Frame)**.** The *frame* of an automaton $\mathcal{A} : \langle \Sigma, Q, \delta, I, F \rangle$ is the tuple $\langle \Sigma, Q, \delta, I \rangle$.

An automaton is called *non-deterministic* whenever I, and $\delta(q, a)$ for all states q and symbols a, have singleton sets as minimal models. In other words, I and $\delta(q, a)$ are equivalent to disjunctive formulas. For example, the FSM of Figure 2.1 can be seen as an alternating automaton which is non-deterministic. A frame is called *universal* if I, and $\delta(q, a)$ for all states q and symbols a, have a unique minimal model. In other words, I and $\delta(q, a)$ are equivalent to conjunctive formulas. A frame is deterministic if it is both non-deterministic and universal, that is if both the initial condition and transition functions correspond to **true**, **false** or a single successor state. In general, a frame is neither universal nor non-deterministic, but fully alternating.

In contrast with finite state machines which have linear runs called paths, the runs of alternating frames are defined as rooted, directed acyclic graphs (DAG).

Since we are interested in the infinite behavior of systems, we will only consider infinite words in the following definitions.

Definition 2.5.5 (Run on a Frame)**.** Given a word $w \in \Sigma^\omega$, a *run* of w on a frame $\mathcal{F} : \langle \Sigma, Q, \delta, I \rangle$ is a DAG (V, E) with nodes $V \subseteq Q \times \mathbb{N}$, such that:

1. $(m, 0) \in V$ for all states m in some minimal model M of I.
2. for every (q, k) in V, E contains an edge $(q, k) \to (q', k + 1)$ for all q' in some minimal model of $\delta(q, w[k])$.

A *trace* of a run is an infinite path in the run.

A non-deterministic frame may admit *multiple different runs* for a given word, but each run contains a *unique trace*. A universal frame admits just *one run* for each word, but this run may contain *multiple traces*. In general a frame admits *multiple runs each with multiple traces*.

Definition 2.5.6 (Specular Frame)**.** Let $\mathcal{F} : \langle \Sigma, Q, \delta, I \rangle$ be a frame. The *specular frame* of $\mathcal{A}$ is $\widetilde{\mathcal{F}} : \langle \Sigma, Q, \widetilde{\delta}, \widetilde{I} \rangle$, where $\widetilde{I}$ is the dual of I and $\widetilde{\delta}$ is the *dual transition function*: $\widetilde{\delta}(q, a)$ is the dual formula of $\delta(q, a)$ for all states q and symbols a.

The *graph* of a frame has Q as a set of nodes and contains an edge $p \to q$ whenever q is in some minimal model of $\delta(p, a)$ for some symbol a. The graphs of a frame and its specular frame are identical, because if q is in some minimal model of $\delta(p, a)$ then q is also in some minimal model of $\widetilde{\delta}(p, a)$. Therefore, a frame admits a trace (a walk in the graph) iff its specular frame also admits the trace.

An *automaton* equips a frame with an *acceptance condition*, which determines whether an infinite sequence of states is accepting. As in the case of FSM, given an infinite sequence of states $\pi : q_0, q_1, q_2 \ldots$ we let $inf(\pi)$ be those states from Q that occur infinitely many times in π. In this work we consider the following acceptance conditions:

***Büchi* :** $F \subseteq Q$ is a set of states and π is accepting when $inf(\pi) \cap F \neq \emptyset$.

***coBüchi* :** $F \subseteq Q$ is a set of states and π is accepting when $inf(\pi) \cap F = \emptyset$.

***parity*:** $F : Q \to \{0 \ldots d\}$ is a map from states to a finite set of natural numbers (to which we refer as *colors*), and π is accepting when $max\{F(q) \mid q \in inf(\pi)\}$ is even.

***Streett*:** $F = \{\langle B_1, G_1 \rangle, \langle B_2, G_2 \rangle, \ldots, \langle B_k, G_k \rangle\}$. π is accepting when for all $1 \leq i \leq k$, if $inf(\pi) \cap B_i \neq \emptyset$ then $inf(\pi) \cap G_i \neq \emptyset$.

***Streett[1]*:** $F = (B, G)$. π is accepting if $inf(\pi) \cap B \neq \emptyset$ then $inf(\pi) \cap G \neq \emptyset$.

***Hesitant*:** $F \subseteq Q$ and $H = \langle (S_0 \ldots, S_k), <, \alpha \rangle$ is a partition of the strongly connected components (SCCs), ordered by $<$ according to reachability in the automaton graph, and α marks each partition as either Büchi or coBüchi. A trace π is accepting when

- $inf(\pi) \subseteq S_i$, S_i is Büchi and $inf(\pi) \cap F \neq \emptyset$, or
- $inf(\pi) \subseteq S_j$, S_j is coBüchi and $inf(\pi) \cap F = \emptyset$.

We use *stratum* to refer to an SCCs of an automaton graph. The stratification of Hesitant automata given by the partition implies that every infinite trace gets trapped in a stratum S_i. Then, the Büchi or coBüchi condition on the stratum determines whether the trace is accepting. We use ABW, AcBW, APW, ASW and AHW to represent Büchi (resp. coBüchi, parity, Streett and Hesitant) alternating automata on words. We use APW$[0, 1, 2]$ for APW that only use colors 0, 1 and 2 and ASW[1] for ASW with only one pair.

When a trace π is accepted according to an acceptance condition F, we write $\pi \in acc(F)$. A run of an alternating automaton is called *accepting* whenever all its traces are accepting. We say that a word w

is in the language of automaton $\mathcal{A}$, and we write $w \in \mathcal{L}(\mathcal{A})$, whenever there is an accepting run for w on $\mathcal{A}$.

Remark. The *size* of an automaton is its number of states (i.e., $|Q|$). When talking about complexity this measure is used.

2.6 Regular Linear Temporal Logic

In this section we recall the main concepts of RLTL, the logic that underlies all this current work.

In 2007, Martin Leuker and César Sánchez [LS07, SL10] proposed Regular Linear-Temporal Logic (RLTL), a logic for the temporal frame, as an extension of LTL [Pnu77, MP95] with constructs based on regular expressions. RLTL extends the expressive power of LTL to all ω-regular languages.

2.6.1 Formal Definition of RLTL

RLTL expressions denote languages over infinite words. We define RLTL in two stages. First, we introduce a variation of regular expressions over finite words, and then—using these—we define regular linear temporal logic to describe langages over infinite words. The syntax of each of these two formalisms consists of an algebraic signature containing a finite collection of constructor symbols. The semantics is given by interpreting these constructors. In particular, the language of RLTL contains no fix-point operators.

Basic Regular Expressions

We first introduce a variation of regular expressions that can define regular languages that do nos contain the empty word. Basic expressions are Boolean combinations of elements from $\mathcal{B}(\mathbb{P})$, for a given set of propositions $\mathbb{P}$, including **true** for $\mathbb{P}$ and **false** for $\varnothing$.

Syntax The language of the regular expressions for finite words is the smalles set closed under:

$$\alpha ::= \alpha + \alpha \;\mid\; \alpha \,;\, \alpha \;\mid\; \alpha * \alpha \;\mid\; p$$

where p ranges over basic expressions. The intended interpretation of the operators $+$, $;$ and $*$ are the standard union, concatenation and binary Kleene-star. We refer to this type of regular expressions as Basic Regular Expressions (*BRE*).

Semantics Our version of regular expressions describes *segments* of infinite words. An infinite word w is a map form ω into Σ (i.e., an element of Σ^ω). A *position* is a natural number. Given an infinite word w and two positions i and j, the tuple (w, i, j) is called a segment of the word w. A *pointed word* is a pair (w, i) formed by a word w and a position i. The semantics is defined inductively as follows. Given a basic expression p, regular expressions x, y and z, and a word w,

(w,i,j)	$\vDash_{RE}$	p	whenever w_i satisfies p and $j = i+1$
(w,i,j)	$\vDash_{RE}$	$x + y$	whenever either $(w,i,j) \vDash_{RE} x$ or $(w,i,j) \vDash_{RE} y$
(w,i,j)	$\vDash_{RE}$	$x \,;\, y$	whenever for some $i \le k < j$, $(w,i,k) \vDash_{RE} x$ and $(w,k,j) \vDash_{RE} y$
(w,i,j)	$\vDash_{RE}$	$x * y$	whenever either $(w,i,j) \vDash_{RE} y$, or for some sequence $(i_0 = i, i_1, \dots i_m < j)$ and for all $k \in \{0,..,m-1\}$ $(w, i_k, i_{k+1}) \vDash_{RE} x$ and $(w, i_m, j) \vDash_{RE} y$

The semantics style used here is more conventional in logic than in automata theory, where regular expressions define sets of finite words. A given regular expression x can be associated with a set of words $\mathcal{L}(x) \subseteq \Sigma^+$, by $v \in \mathcal{L}(x)$ precisely when for some $w \in \Sigma^\omega$, $(vw, 0, |v|) \vDash_{RE} x$. Following this alternative interpretation, our operators correspond to the classical ones and regular expressions define precisely regular sets of non-empty words.

Regular Linear Temporal Logic over Infinite Words

Syntax RLTL is built from regular expressions by using intersection, concatenation of a finite and an infinite expression, and two ternary operators, called the *power* operators. As we will see, the power operators generalize both the LTL constructs and the ω-operator.

The following grammar defines the syntax of RLTL expressions:

$$\varphi ::= \varnothing \;\mid\; \varphi \vee \varphi \;\mid\; \neg\varphi \;\mid\; \alpha\,;\varphi \;\mid\; \varphi|\alpha\rangle\!\rangle\varphi \;\mid\; \varphi|\alpha\rangle\varphi$$

where α ranges over basic regular expressions. The symbol $\vee$ stands for the conventional union of languages (i.e., disjunction in logics and | in semi-extended ω-regular expressions). The symbol ; stands for the conventional concatenation of an expression over finite words and an expression over infinite words.

The operators $\varphi|\alpha\rangle\!\rangle\varphi$ and its weak version $\varphi|\alpha\rangle\varphi$ are the power operators. The power expressions $x|z\rangle\!\rangle y$ and $x|z\rangle y$ (read *x at z until y*, and, respectively, *x at z weak-until y*) are built from three elements: y (the *attempt*), x (the *obligation*) and z (the *delay*). Informally, for $x|z\rangle\!\rangle y$ to hold, either the attempt holds, or the obligation is met and the whole expression evaluates successfully after the delay; in particular, for a power expression to hold the obligation must be met after a finite number of delays. On the contrary, $x|z\rangle y$ does not require the obligation to be met after a finite number of delays. These two simple operators allow the construction of many conventional recursive definitions. For example, the strong until operator of LTL $x\,\mathcal{U}\,y$ can be seen as an attempt for y to hold, and otherwise an obligation for x to be met and a delay of a single step. Similarly, the ω-regular expression x^ω can be interpreted as a weak power operator having no possible escape and a trivially fulfilled obligation, with a delay indicated by x. Conventional ω-regular expressions can describe sophisticated delays with trivial obligations and escapes, while conventional LTL constructs allow complex obligations and escapes, but trivial one-step delays. Power operators can be seen as a generalization of both types of constructs. The completeness of RLTL with respect to ω-regular languages is easily derived from the expressibility of ω-regular expressions. In particular, Wolper's example ("p holds at every other moment") is captured by $p|\mathbf{true}\,;\mathbf{true}\rangle\mathbf{false}$.

Note that the signature of RLTL is, like that of BRE, purely algebraic: the constructors $\vee$ and ; are binary, $\neg$ is unary, the power operators are ternary, and $\varnothing$ is a constant. Even though the symbol ; is overloaded we consider the signatures of BREand RLTL to be disjoint (the disambiguation is clear from the context). The *size* of an RLTL formula is defined as the total number of its symbols.

Semantics

The semantics of RLTL expressions is introduced as a binary relation $\vDash$ between expressions and pointed words, defined inductively. Given two RLTL expressions x and y, a regular expression r and a word w:

(w,i)	$\vDash$	$\varnothing$	never holds.
(w,i)	$\vDash$	$x \vee y$	whenever either $(w,i) \vDash x$ or $(w,i) \vDash y$
(w,i)	$\vDash$	$\neg x$	whenever $(w,i) \nvDash x$, i.e., $(w,i) \vDash x$ does not hold
(w,i)	$\vDash$	$r\,;y$	whenever for some position k, $(w,i,k) \vDash_{RE} r$ and $(w,k) \vDash y$
(w,i)	$\vDash$	$x\|r\rangle\!\rangle y$	whenever $(w,i) \vDash y$ or for some sequence $(i_0 = i, i_1, \ldots i_m)$ $(w,i_k,i_{k+1}) \vDash_{RE} r$ and $(w,i_k) \vDash x$, and $(w,i_m) \vDash y$.
(w,i)	$\vDash$	$x\|r\rangle y$	whenever one of: (*i*) $(w,i) \vDash y$ (*ii*) for some sequence $(i_0 = i, i_1, \ldots i_m)$ $(w,i_k,i_{k+1}) \vDash_{RE} r$ and $(w,i_k) \vDash x$, and $(w,i_m) \vDash y$ (*iii*) for some infinite sequence $(i_0 = i, i_1, \ldots)$ $(w,i_k,i_{k+1}) \vDash_{RE} r$ and $(w,i_k) \vDash x$

The semantics of $x|r\rangle\!\rangle y$ establishes that either the obligation y is satisfied at the point i of the evaluation, or there is a sequence of delays—each determined by r—after which y holds, and x holds after each individual delay. The semantics of $x|r\rangle y$ also allow the case where y never holds, but x always holds after any number of evaluations of r. As with regular expressions, languages can also be associated with

RLTL expressions in the standard form: a word $w \in \Sigma^\omega$ is in the language of an expression x, denoted by $w \in \mathcal{L}(x)$, whenever $(w, 0) \vDash x$. The following lemma follows easily from the definitions:

Lemma 2.6.1. *For every RLTL expressions x and y and BREexpression r:*

- $x|r\rangle\rangle y$ *is semantically equivalent to* $y \vee (x \wedge r\,;\, x|r\rangle\rangle y)$.
- $x|r\rangle y$ *is semantically equivalent to* $y \vee (x \wedge r\,;\, x|r\rangle y)$.

Again, semantic equivalence establishes that both expressions capture the same set of pointed words. Although the semantics of the power operators is not defined using fix point equations, it can be characterized by such equations, similar to the until operator in LTL. A power expression $x|r\rangle\rangle y$ is then characterized to a least fix point, while $x|r\rangle y$ is characterized by a greatest fix-point.

Remark. It should be noted that although RLTL includes complementation it does not allow the use of complementation within regular expressions. It is well-known [Sto74] that emptiness of extended regular expressions (regular expressions with complementation) is not elementary decidable, so this separation is crucial to meet the desired complexity bounds. Similarly, adding intersection to regular expressions—the so-called semi-extended regular expresions—makes the satisfiability problem of similar logics EXPSPACE-complete [Lan07].

The expression $\varnothing$ is needed in RLTL for technical purposes, as a basic case of induction; all other RLTL constructs need some preexisting RLTL expression. The expression $x\,;\neg\varnothing$ that appends sequentially the negation of empty (which corresponds to all pointed words) to a finite expression x serves as a *pump* of the finite models (segments) denoted by x to all infinite words that extend it.

3

Bounded Model Checking for RLTL

In this chapter, we present *bounded model checking* for RLTL and develop a new algorithm based on this technique for the satisfiability and model checking problem for RLTL. Additionally, we define a novel three-valued logic based bounded semantics, which is more precise than the SAT based bounded semantics.

3.1 Introduction

Bounded model checking for LTL was first introduce by Amir Biere in 1999 [BCCZ99]. It is a symbolic model checking technique based on SAT procedures. Its basic idea consist in considering counterexamples of a particular length k and produce a propositional boolean formula that is satisfiable if and only if such a counterexample exists. Due to the depth first search nature of SAT search procedures, bounded model checking for LTL is able to find counterexamples very fast. It also produces counterexamples of minimal length, something very helpful to understand a counterexample more easily. Bounded model checking uses much less space when compared to other symbolic model checking techniques, and it is also fully automatic.

The contributions of this chapter are the followings. We define the *bounded semantics for RLTL* and provide a proof of its correctness. We also present a semantics that is based on a three-valued logic approach. This is a more precise semantics that allows us to distinguish between those cases where a formula is unsatisfiable on a given prefix of a path or is not known whether the formula is satisfied or not along such a prefix, something that cannot be done in the bi-valued semantics. Finally, the main contribution of this chapter is a sound translation from RLTL into propositional formulas for ultimately-periodic words (for convenience, in this thesis we refer to these words as looping-words.)

This chapter is structured as follows. Section 3.2 defines the *bounded semantics* for RLTL, which is an approximation to the unbounded semantics. Section 3.3 presents a bounded semantics that uses three-valued logic. Section 3.4 gives an overview on the concept of *derivatives of regular expressions*, and provides some useful definitions and theorems. Finally, in Section 3.5 the general algorithm to translate RLTL expressions into propositional formulas is provided.

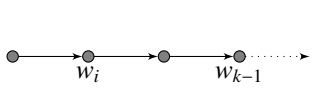
3.2 Bounded Semantics for RLTL over Infinite Words

We start by defining the concept of *k-looping words* . Such words are commonly known as *ultimately-periodic words*, but for convenience and the sake of clarity, throughout this thesis we will refer them as

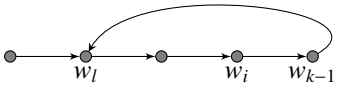
k-looping words, as stated in the definition below.

Definition 3.2.1. For $l \leq k$, we call $w \in \Sigma^\omega$ a (k,l)-looping word if $w = uv^\omega$ with $u = w_0 w_1 \dots w_{l-1}$ and $v = w_l w_{l+1} \dots w_{k-1}$. We simply say that w is a k-looping word if there exists $l \in \mathbb{N}$ with $l \leq k$ such that w is a (k,l)-looping word.

The definition above can be explained graphically as follows.



(a) a no k-looping word



(b) (k,l)-looping word

We now give a first approach to the *bounded semantics* for RLTL, which is an approximation to the unbounded semantics given in Section 2.6. Only expressions that do not contain negation and universal concatenation are considered. This bounded semantics gives us the ability to define the bounded model checking problem for RLTL.

For our purposes, we only consider a finite prefix of a word. To be exact, only the first k letters of a word are used to state if a formula is satisfied along that word. For the case in which the word is a k-looping word we leave the original semantics of RLTL untouched, since the infinite behavior of the word can be reconstructed from the given prefix of length k.

Definition 3.2.2 (Bounded Semantics for Looping Words)**.** Let $k \in \mathbb{N}$, and $w \in \Sigma^\omega$ be a k-looping word. Then an RLTL expression x holds in w with bound k (in symbols $w \vDash_k x$) iff $w \vDash x$.

Definition 3.2.3 (Bounded Semantics for Non-Looping Words)**.** Let $i, k \in \mathbb{N}$ with $i < k$, and let $w \in \Sigma^\omega$ be an infinite word that is not a k-looping word. Then an RLTL expression x holds in w with bound k (in symbols $w \vDash_k x$) iff $(w,i) \vDash_k x$, where

$(w,i) \vDash_k \varnothing$	never holds
$(w,i) \vDash_k \neg\varnothing$	always holds
$(w,i) \vDash_k x \vee y$	whenever either $(w,i) \vDash_k x$ or $(w,i) \vDash_k y$
$(w,i) \vDash_k r; y$	whenever for some position $i \leq j < k$, $(w,i,j) \vDash_{RE} r$ and $(w,j) \vDash_k y$
$(w,i) \vDash_k x\|r\rangle\!\rangle y$	whenever $(w,i) \vDash_k y$ or for some sequence $(i_0 = i, i_1, \dots i_m < k)$ $(w,i_j,i_{j+1}) \vDash_{RE} r$ and $(w,i_j) \vDash_k x$, and $(w,i_m) \vDash_k y$
$(w,i) \vDash_k x\|r\rangle y$	whenever $(w,i) \vDash_k x\|r\rangle\!\rangle y$ holds

Essentially, if w is not a k-looping word, we limit the unbounded semantics to the finite case, i.e., if, starting at position i, no more information than the prefix of length k of w is needed to ensure that the RLTL expression x is satisfied, then we make it also true in the bounded semantics. For the cases where we need more information than the first k letters of w in order to guarantee the satisfiability of the expression x, then we say that x is not satisfiable in the unbounded semantics.

Notice that we cannot define the negation of an RLTL expression in terms of the unbounded semantics due to its intrinsic imprecision. The fact that a formula evaluates to **false** in the bounded semantics for a given prefix does not guarantee the truth of its negation. Cases where uncertainty arises are not managed by this semantics and therefore mapped to **false**. By similar reasons, we cannot make any statement about the infinite behavior of the weak power operator $(x|r\rangle y)$ which can be seen as a power operator $(x|r\rangle\!\rangle y)$ plus the case where y never holds. This behavior cannot be expressed in the bounded semantics for finite prefixes.

The following lemma proof the soundness of our bounded semantics by showing that whenever we can guarantee that an RLTL expression is satisfied by an infinite word within the bounded semantics, it is also the case that this expression is satisfied by the same word in the unbounded semantics.

Lemma 3.2.1. *Let f be an RLTL expression and $w \in \Sigma^\omega$, if $w \vDash_k f$ then $w \vDash f$.*

Proof. If w is a k-looping word the conclusion follows by definition. Let's assume that w is a non-looping word. By induction over the structure of f, we prove the stronger property $(w,i) \models_k f \Rightarrow (w,i) \models f$ for any $i < k$. Let x, y be RLTL expressions and r a regular expression.

- $(w,i) \models_k \varnothing \quad \Leftrightarrow \quad$ **false**
  $\Leftrightarrow \quad (w,i) \models \varnothing$

- $(w,i) \models_k \neg\varnothing \quad \Leftrightarrow \quad$ **true**
  $\Rightarrow \quad (w,i) \not\models \varnothing$
  $\Leftrightarrow \quad (w,i) \models \neg\varnothing$

- $(w,i) \models_k x \vee y \quad \Leftrightarrow \quad (w,i) \models_k x$ or $(w,i) \models_k x$
  $\overset{I.H.}{\Rightarrow} \quad (w,i) \models x$ or $(w,i) \models y$
  $\Rightarrow \quad (w,i) \models x \vee y$

- $(w,i) \models_k r\,;y \quad \Leftrightarrow \quad \exists j < k\,[(w,i,j) \models_{RE} r$ and $(w,i) \models_k y]$
  $\overset{I.H.}{\Rightarrow} \quad \exists j < k\,[(w,i,j) \models_{RE} r$ and $(w,i) \models y]$
  $\Rightarrow \quad \exists j\,[(w,i,j) \models_{RE} r$ and $(w,i) \models y]$
  $\Rightarrow \quad (w,i) \models r\,;y$

- $(w,i) \models_k x|r\rangle\!\rangle y \quad \Leftrightarrow \quad (w,i) \models_k y$ or
  $\exists (i_0 = i, \ldots, i_m < k)\left[(w,i_j,i_{j+1}) \models_{RE} r \text{ and } (w,i_j) \models_k x \text{ and } (w,i_m) \models_k y\right]$
  $\overset{I.H.}{\Rightarrow} \quad (w,i) \models y$ or
  $\exists (i_0 = i, \ldots, i_m < k)\left[(w,i_j,i_{j+1}) \models_{RE} r \text{ and } (w,i_j) \models x \text{ and } (w,i_m) \models y\right]$
  $\Rightarrow \quad (w,i) \models y$ or
  $\exists (i_0 = i, \ldots, i_m)\left[(w,i_j,i_{j+1}) \models_{RE} r \text{ and } (w,i_j) \models x \text{ and } (w,i_m) \models y\right]$
  $\Rightarrow \quad (w,i) \models x|r\rangle\!\rangle y$

- $(w,i) \models_k x|r\rangle y \quad \Leftrightarrow \quad (w,i) \models_k x|r\rangle\!\rangle y$
  $\Rightarrow \quad (w,i) \models x|r\rangle\!\rangle y$
  $\Rightarrow \quad (w,i) \models x|r\rangle y$

□

The last implication in the proof comes from the fact that, in the unbounded semantics, the satisfiability of the formula $x|r\rangle\!\rangle y$ implies the satisfiability of the weaker formula $x|r\rangle y$.

3.3 Bounded Semantics for RLTL using Three-Valued Logic

In this section we define the bounded semantics for RLTL using a three-valued logic. This approach is more precise than using a bi-valued logic since a third value is added to express uncertainty. Then, instead of assigning **false** to those cases where the property is not known to be actually unsatisfiable, we assign them the new *"I don't know"* truth value and we only assign **false** to those cases where the property being checked is guaranteed to be unsatisfiable.

We define our three-valued logic as follows: an expression can be evaluated to any of the values $\top$, $\bot$, or ?, with $\top$ meaning that the expression is **true**, $\bot$ meaning that the expression is **false**, and ? meaning that it is not known whether the expression is **true** or **false**.

Unlike bounded model checking for LTL [BCCZ99]—where a formula is said to be unsatisfiable by a path if not enough information is available when inspecting only a bounded prefix of the path—our approach determines that a prefix of length k of a given infinite word w, does not satisfies a property x (symbolically $(w,0) \models_k^{\bot} x$) if and only if the formula is not satisfied by the entire infinite word (in symbols $w \models x$). In other words, in our previous bounded semantics, as in [BCCZ99], for a given prefix

of length k we can only answer whether it satisfies a formula or not. If the answer is *yes*, we have a witness. Otherwise, if the answer is *no*, we have to increment the k and check again the validity of the formula against the longer prefix because a negative answer also includes all the cases where the formula is not known to be valid. On the other hand, with our new setting, we only increment the k and perform a new test when we obtain a *don't know* answer.

The definition of the bounded semantics is made by fixing the k (i.e., the length of the prefix of the given word w) and distinguishing whether w is a k-looping word or not. If so, we apply the original unbounded semantics since all the information needed to establish the satisfiability of the expression is readily available. If w is not a k-looping word, the bounded semantics is defined taking into consideration the structure of the RLTL expression x, the length of the prefix of w to be inspected and the position $i \leq k$ where x has to be evaluated.

We first extend the semantics for basic regular expressions in order to be able to express uncertainty when the segment of a given word is partially accepted by a regular expression (i.e., the segment is a prefix of a word in the language of the regular expression). It is worth to note that when the word is a k-looping word, the original bi-valued semantics is applied. Also, the bounded semantics coincides with the unbounded semantics in the sense that whenever a regular expression accepts a segment of a word according to the bounded semantics, the unbounded semantics also establishes that the segment is accepted by the regular expression, and viceversa.

Definition 3.3.1 (Bounded Semantics of Regular Expressions for Looping Words)**.** Let $k \in \mathbb{N}$, r a basic regular expression, and $w \in \Sigma^\omega$ be a k-looping word. For any positions i and j such that $i < j \leq k$, r holds in (w, i, j) (in symbols $(w, i, j) \models^k_{RE} r$) iff $(w, i, j) \models_{RE} r$.

Definition 3.3.2 (Bounded Semantics of Regular Expressions for Non-Looping Words)**.** Let $k \in \mathbb{N}$, r a basic regular expression, and $w \in \Sigma^\omega$ be an infinite word that is not a k-looping word. For any positions i and j such that $i < j \leq k$, r holds in (w, i, j) (in symbols $(w, i, j) \models^k_{RE} r$) iff $(w, i, j) \models^\top_{RE} r$, where

(w, i, j)	$\models^\top_{RE}$	p	whenever $p(w_i)$ and $j = i + 1$
(w, i, j)	$\models^\bot_{RE}$	p	whenever $\neg p(w_i)$ or $j > i + 1$
(w, i, j)	$\models^\top_{RE}$	$r + s$	whenever $(w, i, j) \models^\top_{RE} r$ or $(w, i, j) \models^\top_{RE} s$
(w, i, j)	$\models^\bot_{RE}$	$r + s$	whenever $(w, i, j) \models^\bot_{RE} r$ and $(w, i, j) \models^\bot_{RE} s$
(w, i, j)	$\models^\top_{RE}$	$r\ ;\ s$	whenever for some position $i < n < j$, $(w, i, n) \models^\top_{RE} r$ and $(w, n, j) \models^\top_{RE} s$
(w, i, j)	$\models^\bot_{RE}$	$r\ ;\ s$	whenever one of: (i) for every position $i < n \leq j$, $(w, i, n) \models^\bot_{RE} r$ (ii) for every position $i < n < j$ s.t. $(w, i, n) \models^\top_{RE} r$, $(w, n, j) \models^\bot_{RE} s$
(w, i, j)	$\models^\top_{RE}$	$r * s$	whenever $(w, i, j) \models^\top_{RE} s$ or for some position $i < n < j$ s.t. $(w, i, n) \models^\top_{RE} r$, $(w, n, j) \models^\top_{RE} r * s$
(w, i, j)	$\models^\bot_{RE}$	$r * s$	whenever $(w, i, j) \models^\bot_{RE} s$ and for every position $i < n < j$ s.t. $(w, i, n) \models^\top_{RE} r$, $(w, n, j) \models^\bot_{RE} r * s$

and $(w, i, j) \models^?_{RE} r$ whenever $(w, i, j) \not\models^\top_{RE} r$ and $(w, i, j) \not\models^\bot_{RE} r$.

Definition 3.3.3 (Bounded Semantics for Looping Words)**.** Let $k \in \mathbb{N}$, x an RLTL expression, and $w \in \Sigma^\omega$ be a k-looping word. Then x holds in w with bound k (in symbols, $w \models_k x$) iff $w \models x$.

Definition 3.3.4 (Bounded Semantics for non-looping Words)**.** Let $k \in \mathbb{N}$, x an RLTL expression, and $w \in \Sigma^\omega$ be an infinite word that is not a k-looping word. Then x holds in w with bound k (in symbols, $w \models_k x$) iff $(w, 0) \models^\top_k x$, where

(w, i)	$\models^\top_k$	$\varnothing$	never holds
(w, i)	$\models^\bot_k$	$\varnothing$	always holds
(w, i)	$\models^\top_k$	$x \vee y$	whenever $(w, i) \models^\top_k x$ or $(w, i) \models^\top_k y$
(w, i)	$\models^\bot_k$	$x \vee y$	whenever $(w, i) \models^\bot_k x$ and $(w, i) \models^\bot_k y$

(w,i)	$\models_k^\top$	$\neg x$	whenever $(w,i) \models_k^\bot x$
(w,i)	$\models_k^\bot$	$\neg x$	whenever $(w,i) \models_k^\top x$
(w,i)	$\models_k^\top$	$r\,;x$	whenever for some position $j < k$, $(w,i,j) \models_{RE}^\top r$ and $(w,j) \models_k^\top x$
(w,i)	$\models_k^\bot$	$r\,;x$	whenever one of: (i) for all $j \leq k$, $(w,i,j) \models_{RE}^\bot r$ (ii) for all $j < k$ s.t. $(w,i,j) \models_{RE}^\top r$, $(w,j) \models_k^\bot x$
(w,i)	$\models_k^\top$	$r \cdot x$	whenever one of (i) for all $j \leq k$, $(w,i,j) \models_{RE}^\bot r$ (ii) for all $j < k$ s.t. $(w,i,j) \models_{RE}^\top r$, $(w,j) \models_k^\top x$.
(w,i)	$\models_k^\bot$	$r \cdot x$	whenever for some position $j \leq k$, $(w,i,j) \models_{RE}^\top r$ and $(w,j) \models_k^\bot x$.
(w,i)	$\models_k^\top$	$x\|r\rangle\!\rangle y$	whenever $(w,i) \models_k^\top y$, or for some sequence $(i_0 = i, i_1, \ldots i_m < k)$, $(w,i_j,i_j) \models_{RE}^\top r$ and $(w,i_j) \models_k^\top x$, and $(w,i_m) \models_k^\top y$.
(w,i)	$\models_k^\bot$	$x\|r\rangle\!\rangle y$	whenever $(w,i) \models_k^\bot y$, and either there is no sequence $(i_0 = i, i_1, \ldots i_m < k)$ s.t. $(w,i_j,i_{j+1}) \models_{RE}^\top r$ and $(w,i_j) \models_k^{\top,?} x$, or for any sequence $(i_0 = i, i_1, \ldots i_m < k)$ s.t. $(w,i_j,i_{j+1}) \models_{RE}^\top r$ and $(w,i_j) \models_k^{\top,?} x$, $(w,i_m) \models_k^\bot y$.
(w,i)	$\models_k^\top$	$x\|r\rangle y$	whenever $(w,i) \models_k^\top x\|r\rangle\!\rangle y$ (same for $\bot$).

and $(w,i) \models_k^? x$ whenever $(w,i) \not\models_k^\top x$ and $(w,i) \not\models_k^\bot x$.

Lemma 3.3.1 (Soundness)**.** *Let f be an RLTL expression and $w \in \Sigma^\omega$, then $w \models_k^\top f \Rightarrow w \models f$ and $w \models_k^\bot f \Rightarrow w \not\models f$.*

3.4 Derivatives of Basic Regular Expressions

The notion of a derivative of a regular expression was introduced by Brzozowski [Brz64]. In this section we give an overview of the key concepts on the topic. Additionally, we provide a set of useful lemmas and theorems that are used later in this chapter.

3.4.1 Empty Word and Language Emptiness

We begin by defining a set of properties related to the empty word (λ) and language emptiness.

Definition 3.4.1. Given a regular expression r, we define $\delta(r)$ to be

$$\delta(r) = \begin{cases} \lambda & \text{if } \lambda \in \mathcal{L}(r) \\ \varnothing & \text{if } \lambda \notin \mathcal{L}(r) \end{cases}$$

For example, $\delta(r^*) = \lambda$ for any regular expression r, and $\delta(p) = \varnothing$ for any $p \in \Sigma$.

Given a regular expression r, it is often of great interest to know whether or not r accepts the empty word. This concept is formalized in the following definition.

Definition 3.4.2 (Empty Word Property)**.** For any regular expression r, we say that r has the empty word property (denoted by $ewp(r)$) if and only if r accepts the empty word, i.e., $ewp(r) \Leftrightarrow \lambda \in \mathcal{L}(r)$.

For a given regular expression, this property can be computed recursively and the next lemma describes how to do it.

Lemma 3.4.1. *Let r be a regular expression over an alphabet Σ. It can be checked if r has the empty word property, recursively, as follows:*

$$
\begin{array}{lcl}
ewp(\lambda) & = & \textbf{true} \\
ewp(p) & = & \textbf{false} \quad \text{for } p = \varnothing \text{ or } p \in \Sigma \\
ewp(x + y) & = & ewp(x) \vee ewp(y) \\
ewp(x\,;\,y) & = & ewp(x) \wedge ewp(y) \\
ewp(x * y) & = & ewp(y)
\end{array}
$$

Basic regular expressions do not have the empty word property by definition.

Lemma 3.4.2. *If $r \in BRE$, then $\delta(r) = \varnothing$.*

Proof. By definition of *BRE*. A basic regular expression does not accept the empty word. $r \in BRE \Leftrightarrow \lambda \notin \mathcal{L}(r) \Leftrightarrow \delta(r) = \varnothing$. □

Another interesting property is the *emptiness property*, which tells whether the language accepted by a given regular expression is empty. The formal definition of this property is provided below.

Definition 3.4.3 (Emptiness)**.** A regular expression r is empty if and only if the accepted language is empty. In symbols, $emp(r) \Leftrightarrow \mathcal{L}(r) = \varnothing$.

We also provide a way of computing this property.

Lemma 3.4.3. *Let r be a regular expression over an alphabet Σ. It can be checked if r is empty, recursively, as follows:*

$$
\begin{array}{lcl}
emp(\varnothing) & = & \textbf{true} \\
emp(p) & = & \textbf{false} \quad \text{for } p = \lambda \text{ or } p \in \Sigma \\
emp(x + y) & = & emp(x) \wedge emp(y) \\
emp(x\,;\,y) & = & emp(x) \vee emp(y) \\
emp(x * y) & = & emp(y)
\end{array}
$$

3.4.2 Derivatives w.r.t a Letter

In the definition below, we provide the most important concept of this section: the notion of a *derivative of a regular expression* with respect to a letter. Intuitively, the derivative of a regular expression r with respect to a letter $p \in \Sigma$ is also a regular expression s such that $\mathcal{L}(s) = \{u \mid pu \in \mathcal{L}(r)\}$, i.e., s accepts all those words that are accepted by r when adding them the prefix p. In the next subsection we will extend this important concept to words.

Definition 3.4.4 (Derivative of a Regular Expression)**.** Given a regular expression r and a letter $p \in \Sigma$, the derivative of r with respect to p is denoted by $\partial_p(r)$ and is defined recursively as follows:

$$
\begin{array}{lcl}
\partial_p(p) & = & \lambda \\
\partial_p(q) & = & \varnothing \quad \text{for } q = \lambda \text{ or } q = \varnothing \text{ or } q \in \Sigma \text{ and } q \neq p \\
\partial_p(x + y) & = & \partial_p(x) + \partial_p(y) \\
\partial_p(x\,;\,y) & = & \partial_p(x)\,;\,y + \delta(x)\,;\,\partial_p(y) \\
\partial_p(x * y) & = & \partial_p(x)\,;\,x * y + \partial_p(y)
\end{array}
$$

For example, if $\Sigma = \{a, b, c, d\}$, and $r = a * b + c\,;\,c$, by successively applying the rules of Definition 3.4.4, it can be easily checked that $\partial_a(r) = a * b$, $\partial_b(r) = \lambda$, $\partial_c(r) = c$, and $\partial_d(r) = \varnothing$.

The function δ preserves the empty word property as stated by the lemma that follows.

Lemma 3.4.4. *For any regular expression r, $ewp(\delta(r)) \Leftrightarrow ewp(r)$.*

Proof. It follows by definition: $ewp(\delta(r)) \Leftrightarrow \lambda \in \mathcal{L}(\delta(r)) \Leftrightarrow \delta(r) = \lambda \Leftrightarrow ewp(r)$ □

Theorem 3.4.5 proves that the definition of a derivative of a regular expression is well defined in the sense that it matches our intuition of what a derivative is. The proof is performed by proving that the words accepted by the derivative are exactly those that would be accepted by the regular expression if the prefix p were added.

Theorem 3.4.5. *Let $p \in \Sigma$. For any regular expression r, $\mathcal{L}(\partial_p(r)) = \{u \mid pu \in \mathcal{L}(r)\}$.*

Proof. The proof follows by induction over r.

Base Cases. We know that $\mathcal{L}(\varnothing) = \varnothing$, and $\partial_p(r) = \varnothing$ for $r = \varnothing$, $r = \lambda$ or $r = q$ with $q \in \Sigma \setminus \{p\}$. On the other hand, if $r = \varnothing$, the set $\{u \mid pu \in \{\varnothing\}\}$ is empty; if $r = \lambda$, then $\{u \mid pu \in \{\lambda\}\} = \varnothing$; if $r = q \in \Sigma \setminus \{p\}$, the set $\{u \mid pu \in \{q\}\}$ has no elements. We conclude that $\mathcal{L}(\partial_p(r)) = \{u \mid pu \in \mathcal{L}(r)\}$ for any of the base cases.

Induction Cases. Let r_1 and r_2 be regular expressions and assume that $\mathcal{L}(\partial_p(r_1)) = \{u \mid pu \in \mathcal{L}(r_1)\}$ and $\mathcal{L}(\partial_p(r_2)) = \{u \mid pu \in \mathcal{L}(r_2)\}$. Then

- $$\begin{aligned}
\mathcal{L}(\partial_p(r_1 + r_2)) &= \mathcal{L}(\partial_p(r_1) + \partial_p(r_2)) \\
&= \{u \mid pu \in \mathcal{L}(r_1)\} \cup \{u \mid pu \in \mathcal{L}(r_2)\} \\
&= \{u \mid pu \in \mathcal{L}(r_1)\} \cup \{u \mid pu \in \mathcal{L}(r_2)\} \\
&= \{u \mid pu \in \mathcal{L}(r_1) \vee pu \in \mathcal{L}(r_2)\} \\
&= \{u \mid pu \in \mathcal{L}(r_1) \cup \mathcal{L}(r_2)\} \\
&= \{u \mid pu \in \mathcal{L}(r_1 + r_2)\}
\end{aligned}$$

- $$\begin{aligned}
\mathcal{L}(\partial_p(r_1 \,;\, r_2)) &= \mathcal{L}(\partial_p(r_1) \,;\, r_2 + \delta(r_1) \,;\, \partial_p(r_2)) \\
&= \mathcal{L}(\partial_p(r_1) \,;\, r_2) \cup \mathcal{L}(\delta(r_1) \,;\, \partial_p(r_2)) \\
&= \{uv \mid u \in \mathcal{L}(\partial_p(r_1)) \wedge v \in \mathcal{L}(r_2)\} \\
&\quad \cup \{uv \mid u \in \mathcal{L}(\delta(r_1)) \wedge v \in \mathcal{L}(\partial_p(r_2))\} \\
&= \{uv \mid u \in \{t \mid pt \in \mathcal{L}(r_1)\} \wedge v \in \mathcal{L}(r_2)\} \\
&\quad \cup \{uv \mid u \in \mathcal{L}(\lambda) \wedge v \in \{t \mid pt \in \mathcal{L}(r_2)\}\} \\
&\quad \cup \{uv \mid u \in \mathcal{L}(\varnothing) \wedge v \in \{t \mid pt \in \mathcal{L}(r_2)\}\} \\
&= \{uv \mid pu \in \mathcal{L}(r_1) \wedge v \in \mathcal{L}(r_2)\} \cup \{v \mid pv \in \mathcal{L}(r_2)\} \\
&= \{uv \mid puv \in \mathcal{L}(r_1 \,;\, r_2)\}
\end{aligned}$$

- $$\begin{aligned}
\mathcal{L}(\partial_p(r_1 * r_2)) &= \mathcal{L}(\partial_p(r_1) \,;\, r_1 * r_2 + \partial_p(r_2)) \\
&= \mathcal{L}(\partial_p(r_1) \,;\, r_1 * r_2) \cup \mathcal{L}(\partial_p(r_2)) \\
&= \{uv \mid u \in \{t \mid pt \in \mathcal{L}(r_1)\} \wedge v \in \mathcal{L}(r_1 * r_2)\} \\
&\quad \cup \{u \mid pu \in \mathcal{L}(r_2)\} \\
&= \{uv \mid pu \in \mathcal{L}(r_1) \wedge v \in \mathcal{L}(r_1 * r_2)\} \\
&\quad \cup \{uv \mid pu \in \mathcal{L}(r_2) \wedge v = \lambda\} \\
&= \{uv \mid pu \in \mathcal{L}(r_1) \wedge v \in \mathcal{L}(r_1 * r_2) \vee pu \in \mathcal{L}(r_2) \wedge v = \lambda\} \\
&= \{uv \mid puv \in \mathcal{L}(r * s)\}
\end{aligned}$$

We conclude that for any regular expression r and letter $p \in \Sigma$, $\mathcal{L}(\partial_p(r)) = \{u \mid pu \in \mathcal{L}(r)\}$. □

3.4.3 Derivatives w.r.t a Word

We now extend the concept of derivative of a regular expression with respect to a letter to the more general case where the letter is replaced by a word.

Definition 3.4.5 (Derivative of a Regular Expression (General Case))**.** Be $w \in \Sigma^*$ a finite word s.t. $w = p_1 p_2 \ldots p_m$ and be r a regular expression. We define the derivative or r with respect to w (denoted by $\partial_w(r)$) as follows:

$$\partial_{p_1 p_2 \ldots p_m}(r) = \partial_{p_m}\left(\partial_{p_1 p_2 \ldots p_{m-1}}(x)\right) \tag{3.1}$$

For completeness, if $w = \lambda$, $\partial_\lambda(r) = r$.

The following theorem shows that the previous definition is correct.

Theorem 3.4.6. *Given a regular expression r and a word $w \in \Sigma^*$, the derivative of r with respect to w is such that $\mathcal{L}(\partial_w(r)) = \{v \in \Sigma^* \mid wv \in \mathcal{L}(r)\}$.*

Proof. Proof follows from definition 3.4.4. □

The next two definitions provide shortcuts to refer to the empty word property and the emptiness property of a derivative of a regular expression.

Definition 3.4.6 (Derivative Empty Word Property)**.** A regular expression r has the derivative empty word property with respect to a word v (denoted $dewp(w, r)$) whenever $\partial_v(r)$ has the empty word property. It is, $dewp(w, r) \Leftrightarrow ewp(\partial_v(r))$.

Definition 3.4.7 (Derivative Emptiness)**.** We say that r is derivatively empty with respect to a word v (denoted by $demp(v, r)$) if and only if $emp(\partial_v(r))$.

A particular result that will help us to prove some of the results is the following.

Lemma 3.4.7. *For any letter $p \in \Sigma$, $\partial_v(p) = \varnothing$ for all $v \in \Sigma^*$ s.t. $|v| > 1$.*

Proof. Let $p \in \Sigma$ and let $v = a_1 \dots a_n \in \Sigma^*$, with $n > 1$. The proof proceeds by induction on the size of v.

Case [$n = 2$]:

$$\begin{aligned} \partial_{a_1a_2}(p) &= \partial_{a_2}(\partial_{a_1}(p)) \\ &= \begin{cases} \partial_{a_2}(\lambda) & \text{if } a_1 = p \\ \partial_{a_2}(\varnothing) & \text{if } a_1 \neq p \end{cases} \end{aligned}$$

Case [$n > 2$]: Suppose that for any $k < n$, $\partial_{a_1 \dots a_k}(p) = \varnothing$. Then,

$$\begin{aligned} \partial_{a_1 \dots a_n}(p) &= \partial_{a_n}(\partial_{a_1 \dots a_{n-1}}(p)) \\ &= \partial_{a_n}(\varnothing) \\ &= \varnothing \end{aligned}$$

This concludes the proof. □

Next, we provide formulas to evaluate the derivative of the concatenation and binary Kleene-star with respect to a given word of arbitrary length. We also show their correctness.

Lemma 3.4.8. *Let $w \in \Sigma^*$ be a word of arbitrary length, and let i and j be positions such that $i < j$. If $r\,;\,s$ is a basic regular expression, then*

$$\partial_{w_{ij}}(r\,;\,s) = \partial_{w_{ij}}(r)\,;\,s + \sum_{n=i+1}^{j-1} \delta(\partial_{w_{in}}(r))\,;\,\partial_{w_{nj}}(s) \tag{3.2}$$

Proof. We prove it by induction on the size of the segment w_{ij}.

Base Case [$j = i + 1$] Since $r\,;\,s$ is a basic regular expression, it can be shown that $\partial_p(r\,;\,s) = \partial_p(r)\,;\,s$ for any $p \in \Sigma$. So, starting from the righthand side, we obtain

$$\begin{aligned} \partial_{w_{i(i+1)}}(r)\,;\,s + \sum_{n=i+1}^{i} \delta(\partial_{w_{in}}(r))\,;\,\partial_{w_{ni}}(s) &= \partial_{w_{i(i+1)}}(r)\,;\,s + \varnothing \\ &= \partial_{w_{i(i+1)}}(r\,;\,s) \end{aligned}$$

Induction Case [$j > i + 1$] Suppose now that for every $l, k \in \mathbb{N}$, with $k - l < j - i$ the following holds:

$$\partial_{w_{lk}}(r\ ;\ s) = \partial_{w_{lk}}(r)\ ;\ s + \sum_{n=l+1}^{k-1} \delta\left(\partial_{w_{ln}}(r)\right)\ ;\ \partial_{w_{nk}}(s) \tag{3.3}$$

Then,

$$\begin{aligned}
\partial_{w_{ij}}(r\ ;\ s) &= \partial_{w_{(j-1)j}}\left(\partial_{w_{i(j-1)}}(r\ ;\ s)\right) \\
&= \partial_{w_{(j-1)j}}\left(\partial_{w_{i(j-1)}}(r)\ ;\ s + \sum_{n=i+1}^{j-2} \delta\left(\partial_{w_{in}}(r)\right)\ ;\ \partial_{w_{n(j-1)}}(s)\right) \\
&= \partial_{w_{(j-1)j}}\left(\partial_{w_{i(j-1)}}(r)\ ;\ s\right) + \sum_{n=i+1}^{j-2} \partial_{w_{(j-1)j}}\left(\delta\left(\partial_{w_{in}}(r)\right)\ ;\ \partial_{w_{n(j-1)}}(s)\right) \\
&= \partial_{w_{ij}}(r)\ ;\ s + \delta\left(\partial_{w_{i(j-1)}}(r)\right)\ ;\ \partial_{w_{(j-1)j}}(s) \\
&\quad + \sum_{n=i+1}^{j-2} \left(\underbrace{\partial_{w_{(j-1)j}}\left(\delta\left(\partial_{w_{in}}(r)\right)\right)\ ;\ \partial_{w_{n(j-1)}}(s)}_{=\varnothing} + \delta\left(\delta\left(\partial_{w_{in}}(r)\right)\right)\ ;\ \partial_{w_{nj}}(s) \right) \\
&= \partial_{w_{ij}}(r)\ ;\ s + \delta\left(\partial_{w_{i(j-1)}}(r)\right)\ ;\ \partial_{w_{(j-1)j}}(s) + \sum_{n=i+1}^{j-2} \delta\left(\partial_{w_{in}}(r)\right)\ ;\ \partial_{w_{nj}}(s) \\
&= \partial_{w_{ij}}(r)\ ;\ s + \sum_{n=i+1}^{j-1} \delta\left(\partial_{w_{in}}(r)\right)\ ;\ \partial_{w_{nj}}(s)
\end{aligned}$$

□

Lemma 3.4.9. *Let $w \in \Sigma^*$ be a word of arbitrary length, and let i and j be positions such that $i < j$. If $r * s$ is a basic regular expression, then*

$$\partial_{w_{ij}}(r * s) = \partial_{w_{ij}}(r)\ ;\ r * s + \partial_{w_{ij}}(s) + \sum_{n=i+1}^{j-1} \delta\left(\partial_{w_{in}}(r)\right)\ ;\ \partial_{w_{nj}}(r * s) \tag{3.4}$$

Proof. We prove (3.4) by induction on the size of the segment w_{ij}.

Base Case [$j = i + 1$] . Starting with the righthand side, the equality is provided as follows:

$$\partial_{w_{i(i+1)}}(r)\ ;\ r * s + \partial_{w_{i(i+1)}}(s) + \sum_{n=i+1}^{i} \underbrace{\delta\left(\partial_{w_{in}}(r)\right)\ ;\ \partial_{w_{ni}}(r * s)}_{=\varnothing} = \partial_{w_{i(i+1)}}(r * s) \tag{3.5}$$

Induction Case [$j > i + 1$] Assuming that (3.4) holds for any segment of length less than $j - i$, the

following shows that the same is true for segments of length $j - i$.

$$
\begin{aligned}
\partial_{w_{ij}}(r * s) &= \partial_{w_{(j-1)j}}\left(\partial_{w_{i(j-1)}}(r * s)\right) \\
&= \partial_{w_{(j-1)j}}\left(\partial_{w_{i(j-1)}}(r) \,;\, r * s + \partial_{w_{i(j-1)}}(s) + \sum_{n=i+1}^{j-2} \delta\left(\partial_{w_{in}}(r)\right) \,;\, \partial_{w_{n(j-1)}}(r * s)\right) \\
&= \partial_{w_{ij}}(r) \,;\, r * s + \delta\left(\partial_{w_{i(j-1)}}(r)\right) \,;\, \partial_{w_{(j-1)j}}(r * s) + \partial_{w_{ij}}(s) \\
&\quad + \sum_{n=i+1}^{j-2} \partial_{w_{(j-1)j}}\left(\delta\left(\partial_{w_{in}}(r)\right) \,;\, \partial_{w_{n(j-1)}}(r * s)\right) \\
&= \partial_{w_{ij}}(r) \,;\, r * s + \partial_{w_{ij}}(s) + \delta\left(\partial_{w_{i(j-1)}}(r)\right) \,;\, \partial_{w_{(j-1)j}}(r * s) \\
&\quad + \sum_{n=i+1}^{j-2} \left(\underbrace{\partial_{w_{(j-1)j}}\left(\delta\left(\partial_{w_{in}}(r)\right)\right) \,;\, \partial_{w_{n(j-1)}}(r * s)}_{=\varnothing} + \delta\left(\delta\left(\partial_{w_{in}}(r)\right)\right) \,;\, \partial_{w_{nj}}(r * s) \right) \\
&= \partial_{w_{ij}}(r) \,;\, r * s + \partial_{w_{ij}}(s) + \sum_{n=i+1}^{j-1} \delta\left(\partial_{w_{in}}(r)\right) \,;\, \partial_{w_{nj}}(r * s)
\end{aligned}
$$

□

An interesting operator is deriv which, for a given word $w \in \Sigma^*$ and a given set X of regular expressions, computes the set of derivatives of every element of X with respect to w. Next we provide its formal definition.

Definition 3.4.8 (Operator deriv)**.** For $w \in \Sigma^*$ and X a set of regular expressions:

$$\mathsf{deriv}_w(X) = \{\partial_w(s) \mid s \in X\}$$

The following least fixed-point computes the set of derivatives that can be obtained by deriving, an arbitrary number of times, a given regular expression r with respect to a word w.

Definition 3.4.9 (Derivatives of a Loop)**.** Let $w \in \Sigma^*$ and r a regular expression. The derivatives of r with respect to w^*, denoted by $\mathsf{DL}^*(w, r)$, are defined as follows:

$$
\begin{aligned}
\mathsf{DL}^0(w, r) &= \{r\} \\
\mathsf{DL}^i(w, r) &= \mathsf{DL}^{i-1}(w, r) \cup \mathsf{deriv}_w\left(\mathsf{DL}^{i-1}(w, r)\right) \\
\mathsf{DL}^*(w, r) &= \mathsf{LFP}\left(\lambda X \cdot \mathsf{deriv}_w(X) \cup \{r\}\right)
\end{aligned}
$$

If we define the relation sR_wt iff $t = \partial_w(s)$ over any set S of regular expressions, $\mathsf{DL}^*(w, r)$ can be seen as the transitive closure of R_w over the set $\{r\}$ (i.e., $\mathsf{DL}^*(w, r) = R_w^*(\{r\})$.)

3.5 Encoding into SAT

In this section we reduce bounded model checking to propositional satisfiability in order to be able to use efficient propositional decision procedures (SAT) [DP60] to perform model checking. Since SAT procedures do not use canonical forms as BDDs [Bry86] do, they do not suffer from the potential state space explosion of BDDs. A number of efficient SAT solvers have been implemented and propositional satisfiability problems of thousand of variables can be handled with these implementations. The PROVE tool [Bor97], and SATO [Zha97] are remarkable examples of such implementations, both based on Stålmarck's Method [SS90], and Davis & Putnam Procedure [DP60] respectively.

We provide a translation from RLTL expressions into SAT formulas for looping words. This translations is sound with respect to the bi-valued bounded semantics. The translation for the three-valued bounded semantics is a work in progress and is not covered in this thesis.

3.5.1 A Translation for Regular Expressions

We start by defining the translation from regular expressions into SAT formulas for segments of infinite words ("${}^{i}[\![\cdot]\!]_{k}^{j}$"). Recall that a segment is simply a finite subword. The following definition provides a way to recursively compute the associated propositional formula for a given segment and a regular expression.

Definition 3.5.1 (Translation of a Basic Regular Expression for a Segment)**.** Let $k, i, j \in \mathbb{N}$, with $i < j \leq k$, and $w \in \Sigma^{\omega}$. For any given basic regular expression r,

$$
\begin{aligned}
{}^{i}[\![\lambda]\!]_{k}^{j} &= j = i \\
{}^{i}[\![p]\!]_{k}^{j} &= p(w_i) \wedge j = i + 1 \\
{}^{i}[\![r + s]\!]_{k}^{j} &= {}^{i}[\![r]\!]_{k}^{j} \vee {}^{i}[\![s]\!]_{k}^{j} \\
{}^{i}[\![r \,;\, s]\!]_{k}^{j} &= \bigvee_{n=i+1}^{j-1} \left({}^{i}[\![r]\!]_{k}^{n} \wedge {}^{n}[\![s]\!]_{k}^{j} \right) \\
{}^{i}[\![r * s]\!]_{k}^{j} &= {}^{i}[\![s]\!]_{k}^{j} \vee \bigvee_{n=i+1}^{j-1} \left({}^{i}[\![r]\!]_{k}^{n} \wedge {}^{n}[\![r * s]\!]_{k}^{j} \right)
\end{aligned}
$$

The following lemma establishes the soundness of the translation with respect to the semantics of basic regular expressions over segments of infinite words.

Lemma 3.5.1 (Soundness)**.** *Let $w \in \Sigma^{\omega}$, $i, j \in \mathbb{N}$ s.t. $i < j$ and r be a basic regular expression. Then, $dewp\,((w, i, j), r) \Leftrightarrow {}^{i}[\![r]\!]_{k}^{j}$.*

Proof. We prove this by induction over the length of the segment $(j - i)$ and the structure of r.

- For $j = i + 1$,

$$dewp\,((w, i, i+1), r) \overset{\text{by def.}}{\Leftrightarrow} ewp\left(\partial_{w_{i(i+1)}}(r)\right)$$

  - For $r = p$, $ewp\left(\partial_{w_{i(i+1)}}(p)\right) \Leftrightarrow p(w_i) \Leftrightarrow {}^{i}[\![p]\!]_{k}^{i+1}$
  - For $r = r_1 + r_2$, suppose that $ewp\left(\partial_{w_{i(i+1)}}(r_1)\right) \Leftrightarrow {}^{i}[\![r_1]\!]_{k}^{i+1}$ and $ewp\left(\partial_{w_{i(i+1)}}(r_2)\right) \Leftrightarrow {}^{i}[\![r_2]\!]_{k}^{i+1}$. Then,

$$
\begin{aligned}
ewp\left(\partial_{w_{i(i+1)}}(r_1 + r_2)\right) &\Leftrightarrow ewp\left(\partial_{w_{i(i+1)}}(r_1) + \partial_{w_{i(i+1)}}(r_2)\right) \\
&\Leftrightarrow ewp\left(\partial_{w_{i(i+1)}}(r_1)\right) \vee ewp\left(\partial_{w_{i(i+1)}}(r_2)\right) \\
&\Leftrightarrow {}^{i}[\![r_1]\!]_{k}^{i+1} \vee {}^{i}[\![r_2]\!]_{k}^{i+1} \\
&\Leftrightarrow {}^{i}[\![r_1 + r_2]\!]_{k}^{i+1}
\end{aligned}
$$

  - For $r = r_1 \,;\, r_2$, it follows that,

$$
\begin{aligned}
ewp\left(\partial_{w_{i(i+1)}}(r_1 \,;\, r_2)\right) &\Leftrightarrow ewp\left(\partial_{w_{i(i+1)}}(r_1) \,;\, r_2\right) \\
&\Leftrightarrow ewp\left(\partial_{w_{i(i+1)}}(r_1)\right) \wedge ewp\,(r_2) \\
&\overset{r_2 \in BRE}{\Leftrightarrow} \textbf{false}
\end{aligned}
\tag{3.6}
$$

    and

$$
\begin{aligned}
{}^{i}[\![r_1 \,;\, r_2]\!]_{k}^{i+1} &\Leftrightarrow \bigvee_{n=i+1}^{i} \left({}^{i}[\![r_1]\!]_{k}^{n} \wedge {}^{n}[\![r_2]\!]_{k}^{i} \right) \\
&\Leftrightarrow \textbf{false}
\end{aligned}
\tag{3.7}
$$

    From equations (3.6) and (3.7) we conclude that $ewp\left(\partial_{w_{i(i+1)}}(r_1 \,;\, r_2)\right) \Leftrightarrow {}^{i}[\![r_1 \,;\, r_2]\!]_{k}^{i+1}$.

- For $r = r_1 * r_2$, assuming that $ewp\left(\partial_{w_{i(i+1)}}(r_2)\right) \Leftrightarrow {}^{i}[\![r_2]\!]_k^{i+1}$, then

$$\begin{aligned} ewp\left(\partial_{w_{i(i+1)}}(r_1 * r_2)\right) &\Leftrightarrow ewp\left(\partial_{w_{i(i+1)}}\left(\partial_{w_{i(i+1)}}(r_1)\ ;\ r_1 * r_2 + \partial_{w_{i(i+1)}}(r_2)\right)\right) \\ &\Leftrightarrow \left(ewp\left(\partial_{w_{i(i+1)}}(r_1) \wedge ewp(r_1 * r_2)\right) \vee ewp\left(\partial_{w_{i(i+1)}}(r_2)\right)\right) \\ &\overset{r\in BRE}{\Leftrightarrow} ewp\left(\partial_{w_{i(i+1)}}(r_2)\right) \end{aligned} \tag{3.8}$$

  On the other hand,

$$\begin{aligned} {}^{i}[\![r_1 * r_2]\!]_k^{i+1} &\Leftrightarrow {}^{i}[\![r_2]\!]_k^{i+1} \vee \bigvee_{n=i+1}^{i}\left({}^{i}[\![r_1]\!]_k^{n} \wedge {}^{n}[\![r_1 * r_2]\!]_k^{i}\right) \\ &\Leftrightarrow {}^{i}[\![r_2]\!]_k^{i+1} \end{aligned} \tag{3.9}$$

  We can conclude, from (3.8), (3.9), and the induction hypothesis that $ewp\left(\partial_{w_{i(i+1)}}(r_1 * r_2)\right) \Leftrightarrow {}^{i}[\![r_1 * r_2]\!]_k^{i+1}$.

  Up to now, we have shown for any basic regular expression r and any unit length segment of a word w (i.e., $j = i + 1$), that $dewp((w, i, i+1), r) \Leftrightarrow {}^{i}[\![r]\!]_k^{i+1}$. Now we proceed by showing that the equality holds even for arbitrarily long segments of a word.

- For $j > i + 1$, suppose that for any j' s.t. $i < j' < j$, $dewp((w, i, j'), r) \Leftrightarrow {}^{i}[\![r]\!]_k^{j'}$. We will show that $dewp((w, i, j), r) \Leftrightarrow {}^{i}[\![r]\!]_k^{j}$. Like before, by definition, it is verified that $dewp((w, i, j), r) \Leftrightarrow ewp\left(\partial_{w_{ij}}(r)\right)$. We then proceed by structural induction over r.

  - For $r = p$. On the one hand, we have $ewp\left(\partial_{w_{ij}}(p)\right) \Leftrightarrow ewp(\varnothing) \Leftrightarrow \mathbf{false}$. On the other hand, ${}^{i}[\![p]\!]_k^{j} \Leftrightarrow p(w_i) \wedge \underbrace{j = i + 1}_{\text{contradiction!}} \Leftrightarrow \mathbf{false}$. Therefore, $dewp((w, i, j), r) \Leftrightarrow {}^{i}[\![p]\!]_k^{j}$.
  - For $r = r_1 + r_2$, suppose that $ewp\left(\partial_{w_{ij}}(r_1)\right) \Leftrightarrow {}^{i}[\![r_1]\!]_k^{j}$ and $ewp\left(\partial_{w_{ij}}(r_2)\right) \Leftrightarrow {}^{i}[\![r_2]\!]_k^{j}$. Then,

$$\begin{aligned} ewp\left(\partial_{w_{ij}}(r_1 + r_2)\right) &\Leftrightarrow ewp\left(\partial_{w_{ij}}(r_1) + \partial_{w_{ij}}(r_2)\right) \\ &\Leftrightarrow ewp\left(\partial_{w_{ij}}(r_1)\right) \vee ewp\left(\partial_{w_{ij}}(r_2)\right) \\ &\Leftrightarrow {}^{i}[\![r_1]\!]_k^{j} \vee {}^{i}[\![r_2]\!]_k^{j} \\ &\Leftrightarrow {}^{i}[\![r_1 + r_2]\!]_k^{j} \end{aligned}$$

  - For $r = r_1\ ;\ r_2$, assume that $\partial_{w_{lm}}(r_1) \Leftrightarrow {}^{l}[\![r_1]\!]_k^{m}$ and $\partial_{w_{lm}}(r_2) \Leftrightarrow {}^{l}[\![r_2]\!]_k^{m}$ for any $l, m \in \mathbb{N}$ s.t $m - l < j - i$. By lemma 3.4.8 we have,

$$\begin{aligned} ewp\left(\partial_{w_{ij}}(r_1\ ;\ r_2)\right) &\Leftrightarrow ewp\left(\partial_{w_{ij}}(r_1)\ ;\ r_2 + \sum_{n=i+1}^{j-1} \partial_{w_{in}}(r_1)\ ;\ \partial_{w_{nj}}(r_2)\right) \\ &\Leftrightarrow \left(ewp\left(\partial_{w_{ij}}(r_1)\right) \wedge ewp(r_2)\right) \\ &\quad \vee \bigvee_{n=i+1}^{j-1}\left(ewp\left(\delta\left(\partial_{w_{in}}(r_1)\right)\right) \wedge ewp\left(\partial_{w_{nj}}(r_2)\right)\right) \end{aligned}$$

    Since $r_2 \in BRE$, $\lambda \notin \mathcal{L}(r_2)$ and the left part of the disjunction is **false**. Also, following Lemma 3.4.4, $ewp\left(\delta\left(\partial_{w_{in}}(r_1)\right)\right) \Leftrightarrow ewp\left(\partial_{w_{in}}(r_1)\right)$. Thus,

$$ewp\left(\partial_{w_{ij}}(r_1\ ;\ r_2)\right) \Leftrightarrow \bigvee_{n=i+1}^{j-1}\left(ewp\left(\partial_{w_{in}}(r_1)\right) \wedge ewp\left(\partial_{w_{nj}}(r_2)\right)\right)$$

    Observe that, by making $n = i$ we end up with the formula $ewp\left(\partial_{w_{i(i+1)}}(r_1)\right) \wedge ewp\left(ewp\left(\partial_{w_{ij}}(r_2)\right)\right)$ which is **false** since $\partial_{w_{i(i+1)}}(r_1) = r_1$ and $r_1 \in BRE$. Then

$$\begin{aligned} ewp\left(\partial_{w_{ij}}(r_1\ ;\ r_2)\right) &\Leftrightarrow \bigvee_{n=i+1}^{j-1}\left(ewp\left(\partial_{w_{in}}(r_1)\right) \wedge ewp\left(\partial_{w_{nj}}(r_2)\right)\right) \\ &\Leftrightarrow \bigvee_{n=i+1}^{j-1}\left({}^{i}[\![r_1]\!]_k^{n} \wedge {}^{n}[\![r_2]\!]_k^{j}\right) \\ &\Leftrightarrow {}^{i}[\![r_1\ ;\ r_2]\!]_k^{j} \end{aligned}$$

- For $r = r_1 * r_2$, as in the previous case, suppose $\partial_{w_{lm}}(r_1) \Leftrightarrow {}^{l}[\![r_1]\!]^{m}_{k}$ and $\partial_{w_{lm}}(r_2) \Leftrightarrow {}^{l}[\![r_2]\!]^{m}_{k}$ for any $l, m \in \mathbb{N}$ s.t $m - l < j - i$. Following lemma 3.4.9,

$$\begin{aligned} ewp\left(\partial_{w_{ij}}(r_1 * r_2)\right) &\Leftrightarrow ewp\left(\partial_{w_{ij}}(r_1)\ ;\ r_1 * r_2 + \partial_{w_{ij}}(r_2) + \sum_{n=i+1}^{j-1} \delta\left(\partial_{w_{in}}(r_1)\right)\ ;\ \partial_{w_{nj}}(r_1 * r_2)\right) \\ &\Leftrightarrow \left(ewp\left(\partial_{w_{ij}}(r_1)\right) \wedge ewp\left(r_1 * r_2\right)\right) \vee ewp\left(\partial_{w_{ij}}(r_2)\right) \\ &\quad \vee \bigvee_{n=i+1}^{j-1} \left(ewp\left(\delta\left(\partial_{w_{in}}(r_1)\right)\right) \wedge ewp\left(\partial_{w_{nj}}(r_1 * r_2)\right)\right) \end{aligned}$$

  Being $r_1 * r_2 \in BRE$, the first conjunction of the righthand side is **false**. Additionally, applying lemma lemma:ewp-delta-x, $ewp\left(\delta\left(\partial_{w_{in}}(r_1)\right)\right) \Leftrightarrow ewp\left(\partial_{w_{in}}(r_1)\right)$. That way we end-up with

$$\begin{aligned} ewp\left(\partial_{w_{ij}}(r_1 * r_2)\right) &\Leftrightarrow ewp\left(\partial_{w_{ij}}(r_2)\right) \vee \bigvee_{n=i+1}^{j-1} \left(ewp\left(\partial_{w_{in}}(r_1)\right) \wedge ewp\left(\partial_{w_{nj}}(r_1 * r_2)\right)\right) \\ &\Leftrightarrow {}^{i}[\![r_2]\!]^{j}_{k} \vee \bigvee_{n=i+1}^{j-1} \left({}^{i}[\![r_1]\!]^{n-1}_{k} \wedge {}^{n}[\![r_1 * r_2]\!]^{j}_{k}\right) \\ &\Leftrightarrow {}^{i}[\![r_1 * r_2]\!]^{j}_{k} \end{aligned}$$

We conclude that for any infinite word $w \in \Sigma^{\omega}$, positions $i < j$, and any basic regular expression r, it is verified that $dewp\left((w, i, j), r\right) \Leftrightarrow {}^{i}[\![r]\!]^{j}_{k}$. □

We now generalize the translation of basic regular expressions to deal with looping words by defining how to translate a match in a loop ("${}^{i}_{l}[\![\cdot]\!]^{j}_{k}$"). That is, having an infinite word $w = w_{0l}\left(w_{lk}\right)^{\omega}$, we establish precisely when the regular expression matches, respectively, the segment w_{ij} or the set of segments $w_{ik}\left(w_{lk}\right)^{*} w_{lj}$ depending on whether or not $i < j$.

Definition 3.5.2 (Translation of a Regular Expression for a Loop)**.** Let $k, l, i, j \in \mathbb{N}$ s.t. $l < i, j \leq k$ and $i \neq k$, and be $w \in \Sigma^{\omega}$ a (k, l)-looping word. For any given regular expression r,

$${}^{i}_{l}[\![r]\!]^{j}_{k} = {}^{i}[\![r]\!]^{j}_{k} \vee \bigvee_{s \in \mathsf{DL}^{*}\left(\partial_{w_{ik}}(r)\right)} {}^{l}[\![s]\!]^{j}_{k}.$$

The following lemma shows that this translation is well defined.

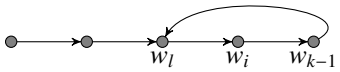
Lemma 3.5.2 (Soundness)**.** *Let $k, l, i, j \in \mathbb{N}$ s.t. $l \leq i, j \leq k$ and $i \neq k$, and be $w \in \Sigma^{\omega}$ a (k, l)-looping word. Then*

$${}^{i}_{l}[\![r]\!]^{j}_{k} \Leftrightarrow \mathcal{L}(w_{ij} + w_{ik}\ ;\ w_{lk} * w_{lj}) \cap \mathcal{L}(r) \neq \varnothing.$$

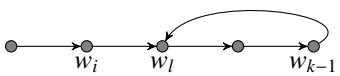
3.5.2 A Translation for RLTL Formulas

We now define the translation from RLTL expressions into propositional formulas. The translation is performed using the previously defined translations for basic regular expressions.

Given a (k, l)-looping word, if a position i is such that $l \leq i \leq k$, we say that i is inside the loop. Otherwise, we say that i is outside the loop. Graphically, the case where i is inside the loop could be depicted as



and the case where i is outside the loop as



Depending on whether or not the starting position of the segment being translated is inside the loop, we have two different translations of the temporal formula x. Definition 3.5.3 provides the translation

for the case in which the starting position is inside the loop ("${}^{i}_{l}[\![\cdot]\!]^{j}_{k}$"). The translation where the starting position is outside the loop ("${}^{i}_{l}\langle\!\langle\cdot\rangle\!\rangle^{j}_{k}$") is given in Definition 3.5.4.

Definition 3.5.3 (Translation of an RLTL Expression Starting Inside a Loop)**.** Let $l, k, i \in \mathbb{N}$, with $l \leq i < k$, and $w \in \Sigma^{\omega}$ a (k, l)-looping word. For an RLTL expression x,

$$
\begin{aligned}
{}^{i}_{l}[\![\varnothing]\!]_{k} &= \textbf{false} \\
{}^{i}_{l}[\![\neg x]\!]_{k} &= \neg {}^{i}_{l}[\![x]\!]_{k} \\
{}^{i}_{l}[\![x \vee y]\!]_{k} &= {}^{i}_{l}[\![x]\!]_{k} \vee {}^{i}_{l}[\![y]\!]_{k} \\
{}^{i}_{l}[\![r\,;x]\!]_{k} &= \left(\bigvee\nolimits_{j=i+1}^{k-1} {}^{i}[\![r]\!]^{j}_{k} \wedge {}^{j}_{l}[\![x]\!]_{k}\right) \vee \left(\bigvee\nolimits_{j=l}^{k-1} {}^{l}_{l}[\![\partial_{w_{ik}}(r)]\!]^{j}_{k} \wedge {}^{j}_{l}[\![x]\!]_{k}\right) \\
{}^{i}_{l}[\![x|r\rangle\!\rangle y]\!]_{k} &= \mathsf{F}^{k-l}\left({}^{i}_{l}[\![x|r\rangle\!\rangle y]\!]_{k}\right), \text{ where} \\
&\qquad \mathsf{F}^{0}\left({}^{i}_{l}[\![x|r\rangle\!\rangle y]\!]_{k}\right) = \textbf{false}, \text{ and} \\
&\qquad \mathsf{F}^{M}\left({}^{i}_{l}[\![x|r\rangle\!\rangle y]\!]_{k}\right) = {}^{i}_{l}[\![y]\!]_{k} \vee {}^{i}_{l}[\![x]\!]_{k} \wedge \bigvee\nolimits_{j=l+1}^{k-1} \left({}^{i}_{l}[\![r]\!]^{j}_{k} \wedge \mathsf{F}^{M-1}\left({}^{j}_{l}[\![x|r\rangle\!\rangle y]\!]_{k}\right)\right) \\
{}^{i}_{l}[\![x|r\rangle y]\!]_{k} &= \mathsf{G}^{k-l}\left({}^{i}_{l}[\![x|r\rangle y]\!]_{k}\right), \text{ where} \\
&\qquad \mathsf{G}^{0}\left({}^{i}_{l}[\![x|r\rangle y]\!]_{k}\right) = \textbf{true}, \text{ and} \\
&\qquad \mathsf{G}^{M}\left({}^{i}_{l}[\![x|r\rangle y]\!]_{k}\right) = {}^{i}_{l}[\![y]\!]_{k} \vee {}^{i}_{l}[\![x]\!]_{k} \wedge \bigvee\nolimits_{j=l+1}^{k-1} \left({}^{i}_{l}[\![r]\!]^{j}_{k} \wedge \mathsf{G}^{M-1}\left({}^{j}_{l}[\![x|r\rangle y]\!]_{k}\right)\right)
\end{aligned}
$$

Definition 3.5.4 (Translation of an RLTL Expression Starting Outside a Loop)**.** Let $l, k, i \in \mathbb{N}$, with $i \leq l < k$, and $w \in \Sigma^{\omega}$ a (k, l)-looping word. For an RLTL expression x,

$$
\begin{aligned}
{}^{i}_{l}\langle\!\langle\varnothing\rangle\!\rangle_{k} &= \textbf{false} \\
{}^{i}_{l}\langle\!\langle\neg x\rangle\!\rangle_{k} &= \neg {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_{k} \\
{}^{i}_{l}\langle\!\langle x \vee y\rangle\!\rangle_{k} &= {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_{k} \vee {}^{i}_{l}\langle\!\langle y\rangle\!\rangle_{k} \\
{}^{i}_{l}\langle\!\langle r\,;x\rangle\!\rangle_{k} &= {}^{l}_{l}[\![\partial_{w_{il}}(r)\,;x]\!]_{k} \vee \bigvee\nolimits_{j=i+1}^{l-1} \left({}^{i}[\![r]\!]^{j}_{k} \wedge {}^{j}_{l}\langle\!\langle x\rangle\!\rangle_{k}\right) \\
{}^{i}_{l}\langle\!\langle x|r\rangle\!\rangle y\rangle\!\rangle_{k} &= {}^{i}_{l}\langle\!\langle y\rangle\!\rangle_{k} \vee \bigvee\nolimits_{j=i+1}^{l-1} \left({}^{i}[\![r]\!]^{j}_{k} \wedge {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_{k} \wedge {}^{j}_{l}\langle\!\langle x|r\rangle\!\rangle y\rangle\!\rangle_{k}\right) \vee \left({}^{l}_{l}[\![x|\partial_{w_{il}}(r)\rangle\!\rangle y]\!]_{k} \wedge {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_{k}\right) \\
{}^{i}_{l}\langle\!\langle x|r\rangle y\rangle\!\rangle_{k} &= {}^{i}_{l}\langle\!\langle y\rangle\!\rangle_{k} \vee \bigvee\nolimits_{j=i+1}^{l-1} \left({}^{i}[\![r]\!]^{j}_{k} \wedge {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_{k} \wedge {}^{j}_{l}\langle\!\langle x|r\rangle y\rangle\!\rangle_{k}\right) \vee \left({}^{l}_{l}[\![x|\partial_{w_{il}}(r)\rangle y]\!]_{k} \wedge {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_{k}\right)
\end{aligned}
$$

The algorithms starts at the beginning of the word by applying the translation outside the loop and constructs the SAT formula bottom-up.

Definition 3.5.5 (General Translation)**.** For $k \in \mathbb{N}$, x be an RLTL expression, and $w \in \Sigma^{\omega}$ a k-looping word:

$$[\![w, x]\!]_{k} = {}^{0}_{l}\langle\!\langle x\rangle\!\rangle_{k}.$$

The soundness of the translation is stated by the following theorem.

Theorem 3.5.3 (Soundness)**.** *Let $k \in \mathbb{N}$, x and RLTL expression, and $w \in \Sigma^{\omega}$ a k-looping word. Then $[\![w, x]\!]_{k}$ is satisfiable iff $w \vDash_{k} x$.*

4

Efficient Translation from RLTL into Büchi Automata

In this chapter we show how to translate RLTL into strong parity automata (APW) with a particular internal structure, and study the complementation construction for the resulting APW. We also study the translation into non-deterministic Büchi automata (NBW). The experimental results demonstrate the efficiency of our method.

4.1 Introduction

The automata-theoretic approach to model checking reduces this verification problem to automata constructions and automata decision problems. The verification process begins by translating the negation of the formula into an equivalent automaton on infinite words. This automaton accepts all the traces that violate the specification. Then, the automaton is composed with the system description using synchronous product composition, for which a non-emptiness check answers whether the system admits some counter-example trace.

Modernly, specifications are translated into alternating automata because their richer structure enables a direct translation from temporal logics, postponing a potentially exponential blow-up. Another advantage of alternation is the easy dualization (see Muller and Schupp [MS87]) provided by the availability of both conjunctive and disjunctive transition relations. However, to obtain an automaton accepting the complement language of a given automaton, one also needs to complement the acceptance condition (see for example [Tho99]). For LTL one can first translate a formula (e.g., the negation of the specification) into negation normal form by pushing negation to the propositional level, and then use automata with weak acceptance conditions [KV01, GO01]. Extensions of LTL with regular expression, like RLTL, do not have negation normal forms. Hence, a translation of the logical negation operator must be given, precluding the use of weak acceptance conditions.

The classical complementation for the parity condition increments in one unit the color assigned to every state, turning an arbitrary sequence of states from accepting into rejecting (and viceversa). However, if this construction is used to translate the logical negation operator, the total number of colors used in the resulting automaton can grow linearly in the size of the formula. The best known algorithm [DK08] for translating an APW with n states and k colors into a non-deterministic Büchi automaton requires $2^{O(nk\ log\ nk)}$ states. Here, we use a faster complementation construction based on the following intuition: only sequences that *occur* in suffix traces belonging to some SCC must be complemented. This idea enables a translation of RLTL including the negation operator, using only colors 0, 1 and 2, which are equivalent to alternating Streett automaton with one accepting pair (denoted ASW[1]). The translation

proceeds inductively, building at each step a pair of complement automata. Then, inspired by this translation we enrich RLTL with new constructors, including universal sequential composition. The enriched logic has a negation normal form.

Street[1] rankings (see [KV05]) directly allow to translate an ASW[1] into an NBW of size $2^{O(n\ logn)}$. Here, we use again the particular stratified structure of the ASW[1] automata obtained from RLTL expressions. Each stratum in the generated ASW[1] is either Büchi (only colors 1 and 2) or coBüchi (colors 0 and 1), making these automata equivalent to hesitant automata AHW (see [KPV01]). We introduce a notion of stratified ranking and show that for all RLTL operators (except one), the ranking of each state can be statically predetermined. This results produces NBW with size $2^{O(n\ logm)}$ where m is the size of the largest strata that cannot be predetermined. In particular, all LTL operators generate strata of size 1, which result into NBW of size $2^{O(n)}$ when using our method.

4.2 Specular Automata Pairs

In this section we introduce the concept of specular automata pairs and present the most relevant results on them. We also explain how this framework allows us to perform an efficient translation from RLTL into NBW.

Positive Boolean Formulas Every positive boolean formula can be expressed in disjunctive normal form, as disjunction of conjunctions of propositions. Given a positive boolean formula θ there is a dual formula $\widetilde{\theta}$ obtained by switching $\wedge$ and $\vee$, and switching **true** and **false**. Some easy properties of dual formulas are:

Proposition 4.2.1 (Duals)**.** *For every θ and $\widetilde{\theta}$, and for every $M \in Mod(\theta)$:*

1. *For every $M' \in Mod(\widetilde{\theta})$, $M \cap M' \neq \varnothing$.*
2. *Let $q \in M$. There is an M' in $Mod(\widetilde{\theta})$ with $q \in M'$.*

For example, if $\theta_1 = (q_1 \vee q_2) \wedge q_3$, then its dual is $\widetilde{\theta_1} = (q_1 \vee q_2) \wedge q_3$, or equivalently in disjunctive normal form $\widetilde{\theta_1} = (q_1 \wedge q_3) \vee (q_2 \wedge q_3)$. The minimal models of $\widetilde{\theta_1}$ are $\{q_1, q_3\}$ and $\{q_2, q_3\}$.

A *choice function* is a map f that chooses for a model M of θ an element of M, i.e., $f : Mod(\theta) \to \mathcal{X}$ such $f(M) \in M$. Some interesting properties of choice functions follow:

Proposition 4.2.2 (Choice Functions)**.** *Let θ be a formula and $\widetilde{\theta}$ its dual. Then*

1. *If f is a choice function for θ, then $Img\ f \in Mod(\widetilde{\theta})$.*
2. *If $M \in mod(\theta)$ then there is a choice function f of $\widetilde{\theta}$ such that $Img\ f = M$.*

Proof. We prove 4.2.2.1 first. Consider θ in disjunctive normal form. Each child subexpression of the root expression corresponds to a conjunction of states that form a model. The choice function f chooses one state from each model of θ. Expressing $\widetilde{\theta}$ dualy, each child subexpression of θ is a disjunction of the corresponding set of states. Hence, the element that f chooses in each child satisfies the corresponding disjunction, and

$$Img\ f = \bigcup_{M \in Mod(\theta)} f(M)$$

is a model of $\widetilde{\theta}$.

We now show 4.2.2.2. Let M be a minimal model of θ. Consider an arbitrary choice function f for $\widetilde{\theta}$ with:

$$f(M') = q \ \text{ for some } \ q \in M \cap M'.$$

By Prop. 4.2.1.1 for any M' there is one such q, so f is well defined, and by construction $Img\ f \subseteq M$. By Prop. 4.2.2.1 $Img\ f$ is a model of θ, and since M is a minimal model it has no proper sub-model, so $Img\ f = M$. □

Definition 4.2.1. Two automata $\mathcal{A} : \langle \Sigma, Q_A, \delta_A, I_A, F_A \rangle$ and $\mathcal{B} : \langle \Sigma, Q_B, \delta_B, I_B, F_B \rangle$ over the same alphabet are specular pairs whenever:

1. their frames are specular (i.e., $Q_B = Q_A$, $\delta_B = \delta_A$, and $I_B = \widetilde{I_A}$), and

2. for all paths π in the frame graph:

$$\pi \in acc(F_A) \text{ if and only if } \pi \notin acc(F_B).$$

One easy way of obtaining a specular automaton $\mathcal{B}$ from a given parity automaton $\mathcal{A}$ is to create the specular frame by dualizing the initial condition and transition functions, and letting $F_B(q) = F_A(q) + 1$ for every state q. Since the order between the colors of two states is preserved, the maximal element q_{max} for a given path will be the same in both automata. Hence, $F_A(q_{max})$ is even precisely when $F_B(q_{max})$ is odd, and $\mathcal{B}$ is a specular automaton of $\mathcal{A}$. This construction is well-known for complementing alternating parity automata.

However, it is possible in many cases to exploit the particular structure of $\mathcal{A}$ to define lower values for F_B, since one only needs to consider those traces that can occur in runs of $\mathcal{A}$.

4.3 Automata and Games

We refer the reader to Harmer's notes [Har] for a detailed introduction to game theory.

We show now that specular automata accept complement languages, using game theory. From a given automaton $\mathcal{A}$ and a word w, we create a parity game called a *word game* as a tuple $\mathbf{G}(\mathcal{A}, w) : \langle V_A, V_P, E_A, E_P, f\rangle$ where:

$$\begin{aligned}
V_A &= Q \times \omega \\
V_P &= \{(M, q, i) \mid M \in Mod(\delta(q, w[i]))\} \cup \{(M, \cdot, 0) \mid M \in Mod(I)\} \\
E_A &= (q, i) \to (M, q, i) \text{ for each } M \in Mod(\delta(q, w[i])) \\
E_P &= (M, q, i) \to (q', i+1) \text{ for } q' \in M \\
f &: V \to \{0 \dots d\}
\end{aligned}$$

The game is played by two players: *Automaton* (A) and *Pathfinder* (P). The set of positions $V = V_A \cup V_P$ is partitioned into positions in which A plays and those in which P plays. The game begins by A choosing a model of I, which determines the initial position $(M, \cdot, 0)$ (here $\cdot$ represents an irrelevant state). The legal moves of the game are captured by the relation $E = E_A \cup E_P$ which correspond to A choosing a model from a V_A position, and P choosing the next successor from a given model from a V_P position. A *play* is an infinite sequence of positions $\pi : V_0 v_0 V_1 v_1 \dots$ with V_0 being an initial position, v_i obtained from V_i by a P move, and V_{i+1} obtained from v_i by an A move. The map f determines the outcome of a play. We define the *trace* of a play $\pi : V_0 v_0 V_1 v_1 \dots$ as the sequence of states $trace(\pi) : p_0 p_1 \dots$ obtained by projecting the first component of the V_P positions of the play (i.e., $v_i = (p_i, i)$). The following follows directly from the definition:

Proposition 4.3.1. *Every trace of a play of $\mathbf{G}(\mathcal{A}, w)$ is also a trace of some run of $\mathcal{A}$ on w.*

As for parity automata the outcome of a play is determined by the highest color that is seen infinitely often in the play. Player A wins play π whenever:

$$max\{f(q) \mid q \in inf(trace(\pi))\} \text{ is even}$$

Otherwise, P wins play π. A strategy for player A is a map $\rho_A : (V^* V_A \cup \epsilon) \to V$, that maps histories of positions into moves. Here, ϵ denotes the empty sequence of positions, to let player A choose an initial state in the game. A memoryless strategy simply takes into account the last position: $\rho_A : V_A \cup \epsilon \to V$. Since parity games are memoryless determined it is enough to consider memoryless strategies. Similarly, a strategy for player P is a map $\rho_P : V_P \to V$. A play $\pi : V_0 v_0 V_1 v_1 \dots$ is played according to strategy ρ_A whenever the initial position is $V_0 = \rho_A(\epsilon)$ and all moves of A are played according to it $V_i = \rho_A(v_i)$. A strategy ρ_A is winning for player A whenever all plays played according to ρ_A are winning for A. Memoryless determinacy of parity games guarantees that either player A has a memoryless winning strategy or player P has a memoryless winning strategy. We say that π is a $G \cdot \rho_A$ play whenever π is played in G according to ρ_A.

We restrict our attention to strategies for A that choose minimal models, and strategies for P that are proper choice functions. This is not a drastic restriction. Clearly, if there is a winning strategy for A that does not choose a minimal model, then any strategy that chooses a smaller minimal model is also winning. This is because the set of plays is reduced, and all plays in the unrestricted set are winning for A. Similarly, if ρ_P is a winning strategy for P, then restricting its moves to be a proper choice function (by restricting the image) also gives a winning strategy. In both cases, the set of successor moves is restricted but still confined within winning regions. This lemma is essentially Prop. 2 from [Tho99], where complementation of weak alternation automata by dualization is studied.

Lemma 4.3.2. *$w \in \mathcal{L}(\mathcal{A})$ if and only if A has a winning strategy in $\mathbf{G}(\mathcal{A}, w)$.*

Proof. Assume $w \in \mathcal{L}(\mathcal{A})$ and let $\sigma : (V_\sigma, E_\sigma)$ be a successful run of w on $\mathcal{A}$. We first build a strategy ρ_A for A on $\mathbf{G}(\mathcal{A}, w)$ and then show that ρ_A is winning:

$$\begin{aligned} \rho_A(\epsilon) &= (M_0, \cdot, 0) \qquad \text{with } M_0 = \{q \mid (q,0) \in V_\sigma\} \\ \rho_A(q,i) &= (M, q, i+1) \qquad \text{with } M = \{q' \mid (q,i) \to (q', i+1) \in E_\sigma\} \end{aligned}$$

The set M in $(M, q, i+1)$ is a model of $\delta(q,i)$ because σ is a run. For positions (q,i) that do not appear in the run σ, the strategy $\rho_A(q,i) = (M, q, i+1)$ can assign any model M in $Mod(\delta(q, w[i]))$. This model is not relevant because no play played according to ρ_A will visit these states. Consider now an arbitrary play $\pi : V_0 v_0 V_1 v_1 \ldots$ of $\mathbf{G}(\mathcal{A}, w)$ played according to ρ_A. We show by induction that $trace(\pi) : p_0 p_1 \ldots$ is a trace of σ.

- *base*: By construction M_0 is the set of initial positions of σ. Since p_0, chosen by player P, is $v_0 \in M_0$, then v_0 is a prefix of a trace of run σ.
- *induction step*: assume $p_0 \ldots p_i$ is a prefix of some trace in σ, so (p_i, i) is in V_σ. Hence, $\rho_A(p_i, i) = (M, p_i, i+1)$ for M being the set of successors of (p_i, i) in E_σ. Consequently $p_{i+1} = (q, i+1)$ for some $(p_i, i) \to (p_{i+1}, i+1)$ in E_σ, so $v_0 \ldots v_i v_{i+1}$ is a longer prefix of a trace of run in σ.

This shows that $trace(\pi)$ is a trace of the run σ. Now, since σ is a successful run all its traces must be accepting, and then:

$$max\{F(q) \mid q \in inf(trace(\pi)\} \text{ is even,}$$

which shows that ρ_A is a winning strategy for $\mathbf{G}(\mathcal{A}, w)$.

We now show the other direction: we start from a winning strategy ρ_A for A in $\mathbf{G}(\mathcal{A}, w)$ and show that there is a successful run σ of w on $\mathcal{A}$. Let $(M, \cdot, 0) = \rho_A(\epsilon)$. Then we let V_σ contain $(q, 0)$ for all $q \in M$. Note that M is a minimal model of I. Now, consider an arbitrary position (q,i) and let $(M, q, i+1)$ be $\rho_A(q,i)$. We add to E_σ all pairs of the form $(q,i) \to (q', i+1)$ for all $q' \in M$. We have to show that σ is successful run. We show by induction that all traces of σ correspond to plays in $\mathbf{G}(\mathcal{A}, w)$ played according to ρ_A. For the base case $(q, 0)$ is the initial state of the trace. By construction $(q,0) \in \rho_A(\epsilon)$ so $(q,0)$ is a possible choice of player P, and consequently a play prefix. For the inductive case, assume that trace prefix $(q_0, 0) \ldots (q_i, i)$ is a play prefix, and let $(q_i, i) \to (q_{i+1}, i+1)$ be in E_σ. By construction $\rho_A(q_i, i)$ contains position $(q_{i+1}, i+1)$ so player P can again move to it. This shows that the arbitrary trace of σ correspond to a play played according to ρ_A. □

4.3.1 Specular Pairs and Complementation

We show now that specular automata accept complement languages. In the rest of the section we let $\mathcal{A}$ and $\widetilde{\mathcal{A}}$ be a specular automata pair, w be a word and $G : \mathbf{G}(\mathcal{A}, w)$ and $\widetilde{G} : \mathbf{G}(\widetilde{\mathcal{A}}, w)$ be the corresponding word games. First we need some preliminary definitions.

Definition 4.3.1. We say that strategies ρ_A (for A in G) and $\widetilde{\rho}_P$ (for P in $\widetilde{G}$) are duals whenever both:

- for every $G \cdot \rho_A$ play π there is a $\widetilde{G} \cdot \rho_P$ play $\widetilde{\pi}$ s.t. $trace(\widetilde{\pi}) = trace(\pi)$.
- for every $\widetilde{G} \cdot \rho_P$ play $\widetilde{\pi}$ there is a $G \cdot \rho_A$ play π s.t. $trace(\widetilde{\pi}) = trace(\pi)$.

Theorem 4.3.3 (Dual Strategies)**.** *The following holds:*
(1) For every strategy ρ_A for A in G, there is a dual strategy $\widetilde{\rho}_P$ for P in $\widetilde{G}$.
(2) For every ρ_P for P in G, there is a dual strategy $\widetilde{\rho}_A$ for A in $\widetilde{G}$.

Proof. We prove the two statements separately:

(1) Let ρ_A be a strategy for A in G. This strategy ρ_A is characterized by

$$\begin{array}{rl} \rho_A(\epsilon) = (M_0, \cdot, 0) & \text{where } M_0 \in mod(I) \\ \rho_A((q,i)) = (M, q, i+1) & \text{where } M \in mod(\delta(q, w[i])) \end{array}$$

By Prop. 4.2.2.1 there are choice functions satisfying

$$\begin{array}{rr} f_{M_0} : Mod(\widetilde{I}) \to Q & Img\ f_{M_0} = M_0 \\ f_{\langle M,q,a\rangle} : Mod(\widetilde{\delta}(q,a)) \to Q & Img\ f_{\langle M,q,a\rangle} = M \end{array}$$

Moreover, these functions are proper choice functions. We now define the dual strategy $\widetilde{\rho}_P$ for P in $\widetilde{G}$ as follows:

$$\begin{array}{rl} \widetilde{\rho}_P((N_0, \cdot, 0)) & = (f_{M_0}(N_0), 0) \\ \widetilde{\rho}_P((N, q, i+1)) & = (f_{\langle M,q,a\rangle}(N), q, i+1) \end{array}$$

where M is the move of A in G from (q,i): $\rho_A(q,i) = (M, q, i+1)$, and $a = w[i]$. Our choice of choice functions $f_{\langle M,q,a\rangle}$ guarantees that for every move of player P from M, there is a move for player A in $\widetilde{G}$ that, when followed by $f_{\langle M,q,a\rangle}$ results in the same state. The properties of f_{M_0} and $f_{\langle M,q,a\rangle}$ ensure that the strategy $\widetilde{\rho}_P$ is proper.

We are ready to show that for every $G{\cdot}\rho_A$ play there is a $\widetilde{G}{\cdot}\rho_P$ play with the same trace, and vice-versa.

"$\Rightarrow$" Consider an arbitrary $G \cdot \rho_A$ play $\pi : V_0 v_0 V_1 v_1 \ldots$, and let $\rho_A(\epsilon) = (M_0, \cdot, 0)$ and $\rho_A(v_i) = (M_{i+1}, q_i, i+1)$. We use q_i for $v_i = (q_i, i)$. Note that $q_{i+1} \in M_{i+1}$ because all moves of player P in π are legal moves. We create the $\widetilde{G} \cdot \rho_P$ play $\widetilde{\pi} : \widetilde{V_0}, \widetilde{v_0}, \widetilde{V_1}, \widetilde{v_1} \ldots$ as follows:

- $\widetilde{V_0} = (N_0, \cdot, 0)$ where N_0 is such that $f_{M_0}(N_0) = q_0$. One such N_0 exists since $Img\ f_{M_0} = M_0$ and $q_0 \in M_0$ (recall that $(q_0, 0)$ is the result of a move of P in G from $(M_0, \cdot, 0)$).
- From (q_i, i), player A chooses in $\widetilde{G}$ the position $(N_{i+1}, q_i, i+1)$, where N_{i+1} is chosen such that $f_{\langle M_{i+1},q,w[i]\rangle} = q_{i+1}$.

By induction, we show that $v_i = \widetilde{v_i}$. First, $\widetilde{v_0} = \widetilde{\rho}_P((N_0, \cdot, 0)) = (f_{M_0}(N_0), 0) = (q_0, 0) = v_0$. Now, assume that for some i, $v_i = \widetilde{v_i}$. Then, $\widetilde{V_i} = (N_{i+1}, q_i, i+1)$, and $V_i = \rho_A(q_i, i) = (M_{i+1}, q_i, i+1)$. Now,

$$\begin{array}{rl} \widetilde{v}_{i+1} = \widetilde{\rho}_P(\widetilde{V_i}) = \widetilde{\rho}_P((N_{i+1}, q_i, i+1)) & = \\ = (f_{\langle M_{i+1},q_i,w[i]\rangle}(N_{i+1}), i+1) & = \\ = (q_{i+1}, i+1) & = \\ = v_{i+1}. & \end{array}$$

Hence, $trace(\pi) = trace(\widetilde{\pi})$.

"$\Leftarrow$" Consider an arbitrary $\widetilde{G} \cdot \rho_P$ play $\widetilde{\pi} : \widetilde{V_0}\widetilde{v_0}\widetilde{V_1}\widetilde{v_1} \ldots$, and let q_i and N_i be such that:

$$\widetilde{v_i} = (q_i, i) \qquad \widetilde{V_0} = (N_0, \cdot, 0) \qquad \widetilde{V}_{i+1} = (N_{i+1}, q_i, i+1)$$

Since $\widetilde{\pi}$ is a $\widetilde{G} \cdot \rho_P$ play, it satisfies that

$$\widetilde{v}_{i+1} = \widetilde{\rho}_P(\widetilde{V}_{i+1}) = (f_{\langle M_{i+1},q_i,w[i]\rangle}(N_{i+1}), i+1)$$

where M_i is obtained from $\rho_A(q_i, i) = (M_{i+1}, i+1)$. Now, we define the play $\pi : V_0 v_0 V_1 v_1 \ldots$ as follows. First the move for A is played according to ρ_A:

$$V_0 = \rho_A(\epsilon) = (M_0, \cdot, 0) \qquad V_{i+1} = \rho_A(v_i)$$

Then, we let the moves of P be:

$$v_0 = (q_0, 0) \qquad v_{i+1} = (q_{i+1}, i+1)$$

We only need to show that these moves for P are legal. First, $q_0 = f_{M_0}(N_0)$, and since $Img\ f_{M_0} = M_0$ it follows that $q_0 \in M_0$, so moving from V_0 into v_0 is a legal move.

Moreover, $(q_{i+1} = f_{\langle M_{i+1},q_i,w[i]\rangle}(N_i))$. Since $Img\ f_{\langle M_{i+1},q_i,w[i]\rangle} = M_{i+1}$ it follows that $q_{i+1} \in M_{i+1}$, so again moving from V_{i+1} into v_{i+1} is a legal move. By construction, $trace(\pi) = trace(\widetilde{\pi})$ again.

(2) Assume now that ρ_P is a (proper) strategy for P in G. The strategy ρ_P is characterized by

$$\rho_P((M_0, \cdot, 0)) = (q_0, 0) \qquad \rho_P((M, q, i)) = (q_i, i)$$

Since the strategy is proper there are proper choice functions:

$$g_0 : Mod(I) \to Q \qquad g_{q,i} : Mod(\delta(q, w[i])) \to Q$$

with

$$\begin{array}{ll} g_0 : Mod(I) \to Q & Img\ g_0 \in mod(\widetilde{I}) \\ g_{q,i} : Mod(\delta(q, w[i])) \to Q & Img\ g_{q,i} \in mod(\widetilde{\delta}(q, w[i])) \end{array} \tag{4.1}$$

We define the strategy $\widetilde{\rho}_A$ for A in $\widetilde{G}$ as follows:

$$\widetilde{\rho}_A(\epsilon) = Img\ g_0 \qquad \widetilde{\rho}_A((q, i)) = Img\ g_{q,i}$$

By (4.1), $\widetilde{\rho}_A$ is well defined. We show now that $\widetilde{\rho}_A$ and ρ_P are dual strategies. First, consider (q, i) an arbitrary state and (M, q, i) a legal move for player A in G. Player P will move to $(q', i+1) = \rho_P((M, q, i))$ with $q' = g_{q,i}((M, q, i))$. In $\widetilde{G}$, player A will move from (q, i) into $(Img\ g_{q,i}, q, i)$. We let player P move in $\widetilde{G}$ to $(q', i + 1)$, which is legal, since $q' \in Img\ g_{q,i}$. Consider now an arbitrary state (p, i) and the move of A in $\widetilde{G}$: $\widetilde{\rho}_A((p, i)) = (Img\ g_{p,i}, p, i)$, and consider an arbitrary legal move for P, $(p', i + 1)$, hence $p' \in Img\ g_{p,i}$. Consequently, there is an $M \in Mod(\delta(p, w[i]))$ such that $g_{p,i}((M, p, i)) = p'$. Let A choose (M, p, i) as the move from (p, i), which is a legal move. Then, playing from (M, p, i) in G according to ρ_P, the resulting state is $(p', i + 1)$. This shows that ρ_A and $\widetilde{\rho}_P$ are dual strategies.

It is important to note that the moves of the players playing against the strategies are not restricted to follow proper strategies (give minimal models or be proper choice functions). Still, ρ_A is winning precisely whenever ρ_P is. □

The following theorem follows directly from Lemma 4.3.2 and Theorem 4.3.3. This theorem allows to reason about complementation simply by reasoning about traces of two automata with dual frames.

Theorem 4.3.4 (Specular Automata and Complement)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be a specular pair of automata. Then $\mathcal{L}(\mathcal{A}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$.*

Theorem 4.3.4 reduces the proof that two automata with dual frames are complements to checking that the traces that can happen have opposite acceptance. In the next section, we use this result to build an incremental translation, in which we only need to check the new traces added at each step.

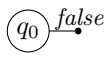
4.4 RLTL into Alternating Automata using Specular Pairs

We present here a translation of RLTL expressions into APW[0, 1, 2] based on Theorem 4.3.4. The main idea is to generate, at each step, a specular automata pair with the first automaton accepting the same language as the expression. By duality, the specular automaton accepts the complement language. Handling logical negation becomes trivial: one simply needs to switch the elements of the pair.

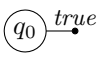
A previous translation of RLTL presented in [SL10] needed n colors instead of 3. Using [KV05, DK08] to translate APW into NBW would produce NBW with $2^{O(n^2 \log n)}$ states for the old translation and $2^{O(n \log n)}$ states for the one presented here. In Section 4.5 below we show how to reduce it further to $2^{O(n \log m)}$ (where m is the size of the largest stratum), and $2^{O(n)}$ for the LTL fragment of RLTL.

The translation is described inductively. For every operator, we show how to compute the specular automata pair, starting from the automata pairs for the sub-expressions. In particular, assume that $(\mathcal{A}_x, \overline{\mathcal{A}_x})$ and $(\mathcal{A}_y, \overline{\mathcal{A}_y})$ are specular pairs for RLTL expressions x and y and that N_r is an NFA for regular expression r. We use $q \to_a F_r$ for "$q \in Q_r$ and $\delta(q, a) \cap F_r \neq \emptyset$," and we use $q \not\to_a F_r$ for "$q \in Q_r$ and $\delta(q, a) \cap F_r = \emptyset$."

Empty: We construct the pair $(\mathcal{A}_\varnothing, \overline{\mathcal{A}_\varnothing})$ with state set $Q = \{q_0\}$. The initial conditions are $I = q_0$ and $\overline{I} = q_0$. The acceptance conditions are $F(q_0) = 0$ and $\overline{F}(q_0) = 0$. The transition relations are $\delta(q_0, \_) = \mathbf{false}$ and $\overline{\delta}(q_0, \_) = \mathbf{true}$. Graphically:



Automaton $\mathcal{A}_\varnothing$ for $\varnothing$.



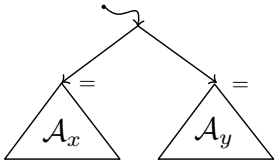
Specular automaton $\overline{\mathcal{A}_\varnothing}$.

Disjunction: The state space of both $\mathcal{A}_{x\vee y} : \langle \Sigma, Q, \delta, I, F\rangle$ and $\overline{\mathcal{A}_{x\vee y}} : \langle \Sigma, Q, \overline{\delta}, \overline{I}, \overline{F}\rangle$ is $Q = Q_x \cup Q_y$. The initial conditions are $I = I_x \vee I_y$ and $\overline{I} = \overline{I_x} \wedge \overline{I_y}$. The transition functions and acceptance condition are:

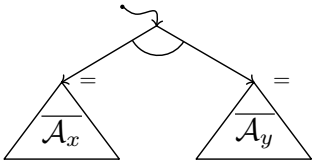
$$\delta(q,a) = \begin{cases} \delta_x(q,a) & \text{if } q \in Q_x \\ \delta_y(q,a) & \text{if } q \in Q_y \end{cases} \qquad \overline{\delta}(q,a) = \begin{cases} \overline{\delta_x}(q,a) & \text{if } q \in Q_x \\ \overline{\delta_y}(q,a) & \text{if } q \in Q_y \end{cases}$$

$$F(q) = \begin{cases} F_x(q) & \text{if } q \in Q_x \\ F_y(q) & \text{if } q \in Q_y \end{cases} \qquad \overline{F}(q) = \begin{cases} \overline{F_x}(q) & \text{if } q \in Q_x \\ \overline{F_y}(q) & \text{if } q \in Q_y \end{cases}$$

Graphically,



Automaton $\mathcal{A}_{x\vee y}$ for $x \vee y$.



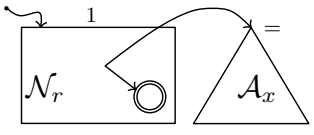
Specular automaton $\overline{\mathcal{A}_{x\vee y}}$.

Sequential: The state space of both $\mathcal{A}_{r;x} : \langle \Sigma, Q, \delta, I, F\rangle$ and $\overline{\mathcal{A}_{r;x}} : \langle \Sigma, Q, \overline{\delta}, \overline{I}, \overline{F}\rangle$ is $Q_r \cup Q_x$. The initial conditions are $I = I_r$ and $\overline{I} = \widetilde{I_r}$. The transition function and acceptance condition are:

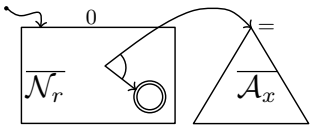
$$\delta(q,a) = \begin{cases} \delta_r(q,a) & \text{if } q \not\to_a F_r \\ \delta_r(q,a) \vee I_x & \text{if } q \to_a F_r \\ \delta_x(q,a) & \text{if } q \in Q_x \end{cases} \qquad \overline{\delta(q,a)} = \begin{cases} \widetilde{\delta_r}(q,a) & \text{if } q \not\to_a F_r \\ \widetilde{\delta_r}(q,a) \wedge I_x & \text{if } q \to_a F_r \\ \delta_x(q,a) & \text{if } q \in Q_x \end{cases}$$

$$F(q) = \begin{cases} 1 & \text{if } q \in Q_r \\ F_x(q) & \text{if } q \in Q_x \end{cases} \qquad \overline{F}(q) = \begin{cases} 0 & \text{if } q \in Q_r \\ \overline{F_x}(q) & \text{if } q \in Q_x \end{cases}$$

Graphically,

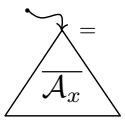


Automaton $\mathcal{A}_{\neg x}$ for $\neg x$.

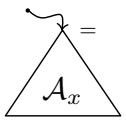


Specular automaton $\overline{\mathcal{A}_{\neg x}}$.

Complementation: Consider now an RLTL sub-expression x, with specular pair $(\mathcal{A}_x, \overline{\mathcal{A}_x})$. Since $(w,i) \vDash \mathcal{A}_x$ if and only if $(w,i) \nvDash \overline{\mathcal{A}_x}$, it follows that $(\overline{\mathcal{A}_x}, \mathcal{A}_x)$ is a specular pair for $\neg x$. Graphically,



Automaton $\mathcal{A}_{r;x}$ for $r\,;\,x$.



Specular automaton $\overline{\mathcal{A}_{r;x}}$.

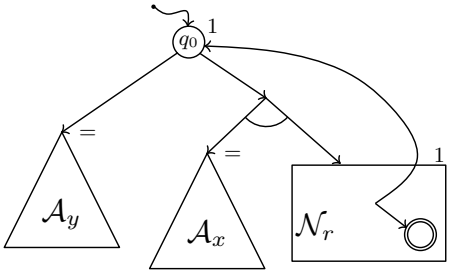
Power: Let q_0 be a fresh state, not present in Q_x or Q_y. The state space of both $\mathcal{A}_{x|r\rangle\!\rangle y} : \langle \Sigma, Q, \delta, I, F\rangle$ and $\overline{\mathcal{A}_{x|r\rangle\!\rangle y}} : \langle \Sigma, Q, \overline{\delta}, \overline{I}, \overline{F}\rangle$ is $Q_x \cup Q_y \cup \{q_0\}$. The initial conditions are $I = q_0$ and $\overline{I} = q_0$. For the transition relation and the acceptance condition:

$$\delta(q_0, \epsilon) = I_y \vee (I_x \wedge I_r) \qquad \overline{\delta}(q_0, \epsilon) = \overline{I_y} \wedge (\overline{I_x} \vee \overline{I_r})$$

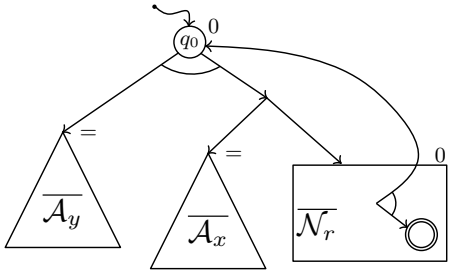
$$\delta(q, a) = \begin{cases} \delta_r(q,a) & \text{if } q \not\rightarrow_a F_r \\ \delta_r(q,a) \vee q_0 & \text{if } q \rightarrow_a F_r \\ \delta_x(q,a) & \text{if } q \in Q_x \\ \delta_y(q,a) & \text{if } q \in Q_y \end{cases} \qquad \overline{\delta}(q, a) = \begin{cases} \widetilde{\delta_r}(q,a) & \text{if } q \not\rightarrow_a F_r \\ \widetilde{\delta_r}(q,a) \wedge q_0 & \text{if } q \rightarrow_a F_r \\ \overline{\delta_x}(q,a) & \text{if } q \in Q_x \\ \overline{\delta_y}(q,a) & \text{if } q \in Q_y \end{cases}$$

$$F(q) = \begin{cases} 1 & \text{if } q = q_0 \\ 1 & \text{if } q \in Q_r \\ F_x(q) & \text{if } q \in Q_x \\ F_y(q) & \text{if } q \in Q_y \end{cases} \qquad \overline{F}(q) = \begin{cases} 0 & \text{if } q = q_0 \\ 0 & \text{if } q \in Q_r \\ \overline{F_x}(q) & \text{if } q \in Q_x \\ \overline{F_x}(q) & \text{if } q \in Q_y \end{cases}$$

Graphically,



Automaton $\mathcal{A}_{x|r\rangle\!\rangle y}$ for $x|r\rangle\!\rangle y$.



Specular automaton $\overline{\mathcal{A}_{x|r\rangle\!\rangle y}}$.

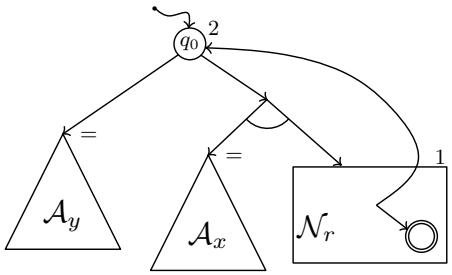
Weak Power: Again, the state space of both $\mathcal{A}_{x|r\rangle y} : \langle \Sigma, Q, \delta, I, F\rangle$ and $\overline{\mathcal{A}_{x|r\rangle y}} : \langle \Sigma, Q, \overline{\delta}, \overline{I}, \overline{F}\rangle$ is $Q_x \cup Q_y \cup \{q_0\}$ for a fresh state q_0. For the initial condition $I = q_0$ and $\overline{I} = q_0$. For the transition relation and acceptance condition:

$$\delta(q_0, \epsilon) = I_y \vee (I_x \wedge I_r) \qquad \overline{\delta}(q_0, \epsilon) = \overline{I_y} \wedge (\overline{I_x} \vee \overline{I_r})$$

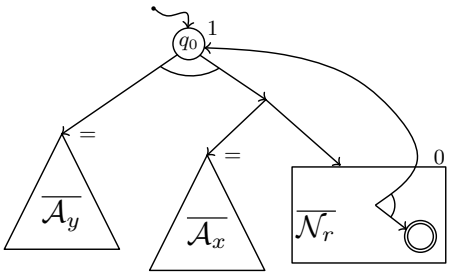
$$\delta(q, a) = \begin{cases} \delta_r(q,a) & \text{if } q \not\rightarrow_a F_r \\ \delta_r(q,a) \vee q_0 & \text{if } q \rightarrow_a F_r \\ \delta_x(q,a) & \text{if } q \in Q_x \\ \delta_y(q,a) & \text{if } q \in Q_y \end{cases} \qquad \overline{\delta}(q, a) = \begin{cases} \widetilde{\delta_r}(q,a) & \text{if } q \not\rightarrow_a F_r \\ \widetilde{\delta_r}(q,a) \wedge q_0 & \text{if } q \rightarrow_a F_r \\ \overline{\delta_x}(q,a) & \text{if } q \in Q_x \\ \overline{\delta_y}(q,a) & \text{if } q \in Q_y \end{cases}$$

$$F(q) = \begin{cases} 2 & \text{if } q = q_0 \\ 1 & \text{if } q \in Q_r \\ F_x(q) & \text{if } q \in Q_x \\ F_y(q) & \text{if } q \in Q_y \end{cases} \qquad \overline{F}(q) = \begin{cases} 1 & \text{if } q = q_0 \\ 0 & \text{if } q \in Q_r \\ \overline{F_x}(q) & \text{if } q \in Q_x \\ \overline{F_x}(q) & \text{if } q \in Q_y \end{cases}$$

Graphically,



Automaton $\mathcal{A}_{x|r\rangle y}$ for $x|r\rangle y$.



Specular automaton $\overline{\mathcal{A}_{x|r\rangle y}}$.

Theorem 4.4.1. *Let φ be an RLTL expression and $\mathcal{A}_\varphi$ be the* ASW[1] *automaton obtained using the construction described in this section. Then, $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A}_\varphi)$.*

The proof of Theorem 4.4.1 is greatly simplified by Theorem 4.3.4 because at every stage automata with dual frame are built and one only needs to reason about the acceptance of traces in the freshly added states. The construction also satisfies two important properties:

1. each stage introduces a new stratum (SCC) that is above all strata from previous stages. In other words, traces that move to the automaton of a sub-expression do not visit the stratum added for the containing expression.
2. The stratum at each stage is decorated only with color 0 (an accepting stratum), only with color 1 (a rejecting stratum), only with colors 0 and 1 (a coBüchi stratum) or only with colors 1 and 2 (a Büchi stratum).

These two observations imply that the automaton has the particular structure of a hesitant automaton AHW. We show in Section 4.5 how to efficiently translate AHW into NBW using a refined version of Streett rankings.

A Universal Sequential Operator. In the previous construction, we observe that the dual automaton for the sequential operator $r\,;\,x$ describes the set of traces in which "*all occurrences of r (if any) are followed by failing occurrences of x*". This observation inspires the introduction of the universal sequential operator $r \cdot x$ with the following semantics:

$(w, i) \vDash_k r \cdot x$ whenever for all k for which $(w, i, k) \vDash_{RE} r$, then $(w, k) \vDash x$.

The translation of $r \cdot x$ is precisely $\overline{\mathcal{A}_{r;\bar{x}}}$ above, and the dual automaton is exactly $\mathcal{A}_{r;\bar{x}}$. Note that the stratum corresponding to r in $\mathcal{A}_{r\cdot x}$ has a universal frame, obtained by dualizing the non-deterministic transition relation of N_r. The following duality laws follows immediately:

$$\neg(r\,;\,x) \equiv r \cdot \neg x \qquad\qquad \neg(r \cdot x) \equiv r\,;\,\neg x \tag{4.2}$$

Universal Power Operators. Similarly, we define new operators $x\|r\rangle\!\rangle y$ and $x\|r\rangle y$, duals of $x|r\rangle y$ and $x|r\rangle\!\rangle y$, respectively. These new operators force repetitions to hold at *all* possible delays, instead of at some possible delay. The semantics are:

$(w, i) \vDash \quad x\|r\rangle\!\rangle y$ whenever $(w, i) \vDash y$ and
for all sequences $(i = i_0, \ldots, i_m)$ with $(w, i_k, i_{k+1}) \vDash_{RE} r$,
either $(w, i_j) \vDash x$ for some $j \leq k$ or $(w, i_{k+1}) \vDash y$, and
for all infinite sequences $(i = i_0, i_1 \ldots)$ with $(w, i_k, i_{k+1}) \vDash_{RE} r$
and $(w, i_k) \vDash y$, there is an m with $(w, i_m) \vDash x$.

$(w, i) \vDash \quad x\|r\rangle y$ whenever $(w, i) \vDash x\|r\rangle\!\rangle y$ or
$(w, i) \vDash y$ and for all natural numbers $k > 0$:
for all j with $(w, i, j) \vDash_{RE} r^k$ then $(w, j) \vDash y$.

The translation of $x\|r\rangle\!\rangle y$ is $\overline{\mathcal{A}_{\bar{x}|r\rangle\bar{y}}}$ (the dual being $\overline{\mathcal{A}_{x\|r\rangle\!\rangle y}} = \mathcal{A}_{\bar{x}|r\rangle\bar{y}}$), and the translation of $x\|r\rangle y$ is the

pair $(\overline{\mathcal{A}_{x|r\rangle\!\rangle y}}, \mathcal{A}_{x|r\rangle\!\rangle y})$. The following laws hold:

$$\neg(x|r\rangle\!\rangle y) \equiv \neg x||r\rangle\neg y \qquad \neg(x||r\rangle y) \equiv \neg x|r\rangle\!\rangle\neg y \tag{4.3}$$

$$\neg(x|r\rangle y) \equiv \neg x||r\rangle\!\rangle\neg y \qquad \neg(x||r\rangle\!\rangle y) \equiv \neg x|r\rangle\neg y \tag{4.4}$$

Finally, $x \wedge y$ is defined with translation $\overline{\mathcal{A}_{\overline{x}\vee\overline{y}}}$, and dual $\mathcal{A}_{\overline{x}\vee\overline{y}}$. The following duality laws (de Morgan laws) hold for the Boolean operators:

$$\neg\neg x \equiv x \qquad \neg(x \vee y) \equiv \neg x \wedge \neg y \qquad \neg(x \wedge y) \equiv \neg x \vee \neg y \tag{4.5}$$

Orienting (4.2)–(4.5) from left to right allows to push logical negation $\neg$ to the propositional level, so RLTL extended with these operators admits a negation normal form. Note that this negation normal form is obtained after the translation by dual pairs. It does not follow immediately that the existence of such a normal form enables a translation into automata with weak acceptance condition, because one has to show translations for the new operators, including essentially all elements of pairs in the translation of RLTL presented above.

4.5 From Stratified ASW[1] into NBW

This section shows how to translate the alternating automata obtained in Section 4.4 into NBW. We first revisit the notion of Streett ranking from [KV05], which in turn is based on the notion of coBüchi ranking [KV04]. Then, we refine rankings to exploit the stratification of the automata obtained as a result of the translation from RLTL. The first observation is that a parity acceptance condition with colors $\{0, 1, 2\}$ corresponds to the Streett condition (B, G) with

$$B = \{q \mid F(q) = 1\} \qquad G = \{q \mid F(q) = 2\}$$

The Streett pair (B, G) forces that if some state marked 1 is visited infinitely often, then some state marked 2 is also visited infinitely often, and hence the trace is accepting. The other possible case is that only states that are not marked either B or G states are visited infinitely often. In this case, the trace is also good for the parity automaton. We first show a general translation of ASW[1] into NBW.

4.5.1 Rankings for ASW[1]

We use $[k]$ as an abbreviation for the set $\{1 \dots k\}$. The following definitions assume a given ASW[1] automaton $\mathcal{A}$ with n states, acceptance condition (B, G), a word $w \in \Sigma^\omega$ and a run $\mathcal{G} : (V, E)$ of $\mathcal{A}$ on w.

Definition 4.5.1. An S[1]-ranking is a function $f : V \to [2n]$ that satisfies:
(i) if $q \in B$ then $f(\langle q, l\rangle)$ is even,
(ii) for every $\langle q, l\rangle \to \langle q', l'\rangle$ in E, $f(\langle q, l\rangle) \geq f(\langle q', l'\rangle)$, unless $q \in G$.

It follows that for every path π on a run DAG $\mathcal{G}$, either π visits infinitely often G states or, after some prefix, condition (*ii*) applies continuously. Hence, since the image of f is bounded, the value of f converges to a value: there is a number l such that, for every $l' > l$, $f(\pi(l')) = f(\pi(l))$. The following definition of odd S[1]-ranking relates the convergence to an odd value with the fact that B states are visited only finitely often.

Definition 4.5.2 (odd S[1]-ranking)**.** An S[1]-ranking is odd whenever, for every path π of $\mathcal{G}$, either
(i) π visits infinitely often G states, or
(ii) f converges to an odd value on π.

Before we prove the main result of this section, that relates accepting runs with the existence of an odd S[1]-ranking, we first need an intermediate result:

Lemma 4.5.1. *Let $\mathcal{G}$ be an accepting run, and let $\mathcal{G}'$ be a non-empty sub-graph of $\mathcal{G}$ with no G vertices and only infinite paths. Then, there is some node in $\mathcal{G}'$ that cannot access any B node.*

Proof. Consider, by contradiction that there is no one such a node in $\mathcal{G}' = (V', E')$, or equivalently, that all vertices in $\mathcal{G}'$ can access a B node:

$$\text{for all } \langle q, l\rangle \in V', \text{ there is some } \langle q', l'\rangle \in V' \text{ with } q' \in B \text{ and } \langle q, l\rangle \to^*_{E'} \langle q', l'\rangle\ .$$

Then, every node can be associated with a B node by a map $\mathit{next}(\langle q, l\rangle)$ that returns one path to a B reachable node (for example, the shortest non-empty path to a B state, and picking the smallest according to some lexicographic order among the shortest ones.) Then using induction define, starting from an arbitrary node $\langle q, l\rangle \in V'$, an infinite path in $\mathcal{G}'$ that visits B nodes infinitely often by concatenating the paths returned by *next*. Let us call π one such path.

Since $\pi(0) = \langle q, l\rangle$ is a node of $\mathcal{G}'$, and consequently a node of $\mathcal{G}$, $\pi(0)$ is reachable from some initial node by point 3 in the definition of a run. Let π_{pre} be a finite path in $\mathcal{G}$ from a node $\langle q_0, 0\rangle \in V$. The path $\pi_{\text{pre}}\pi$ is a trace in $\mathcal{G}$ that visits G nodes finitely often (only nodes in π_{pre} can possibly be G nodes) and B nodes infinitely often in π. This trace contradicts that $\mathcal{G}$ is an accepting run. □

We will use the following notation, for a given sub-graph $\mathcal{G}'$ of a run:

$$\begin{aligned}
\mathit{access}(\mathcal{G}', \langle q, l\rangle) &\stackrel{\text{def}}{=} \{\langle q', l'\rangle \mid \langle q, l\rangle \to^* \langle q', l'\rangle\}\\
\mathit{finite}(\mathcal{G}') &\stackrel{\text{def}}{=} \{\langle q, l\rangle \mid \mathit{access}(\mathcal{G}', \langle q, l\rangle) \text{ is finite}\}\\
\mathit{nobad}(\mathcal{G}') &\stackrel{\text{def}}{=} \{\langle q, l\rangle \mid \mathit{access}(\mathcal{G}', \langle q, l\rangle) \cap B = \emptyset\}\\
\mathit{width}(\mathcal{G}', l) &\stackrel{\text{def}}{=} |\{\langle q, l\rangle \in \mathcal{G}'\}|
\end{aligned}$$

Now we are ready to state and prove the following result, which justifies the construction of the NBW below.

Lemma 4.5.2. *$\mathcal{G}$ is an accepting run iff there is an odd* S[1]*-ranking for $\mathcal{G}$.*

Proof. We prove the two directions separately:

"⇐" Assume there is an odd S[1]-ranking f for $\mathcal{G}$ and let π be an arbitrary trace of $\mathcal{G}$. Since, f is odd, either π visits infinitely many G states, in which case π is accepting, or $f(\pi)$ converges to an odd value. In this second case, there is l such that for all $l' > l$, $f(\pi(l')) = f(\pi(l))$ and $f(\pi(l))$ is odd. By definition of S[1]-ranking (point (i)), $\pi(l')$ cannot be a B state, and consequently π visits only finitely many B states. Hence, π is an accepting trace.

"⇒" Assume $\mathcal{G}$ is an accepting run for $\mathcal{A}$.

- **Initial Stage** The construction of f starts by removing from $\mathcal{G}$ all G vertices. Let V_G be $\{\langle q, l\rangle \mid q \in G\}$, then $f(\langle q, l\rangle) = 0$ for all $\langle q, l\rangle \in V_G$. Also, let $V_0 = \mathit{finite}(\mathcal{G} \setminus V_G)$, we let $f(\langle q, l\rangle = 0$ for all $\langle q, l\rangle \in V_0$. Also $\mathcal{G}_0 = \mathcal{G} \setminus (V_G \cup V_0)$, which contains the original graph except the G nodes, and every node that reaches G nodes in all its outgoing paths.
- **Incremental Stage** The algorithm proceeds in at most n rounds, performing the following two operations in each round k. The round begins with subgraph $\mathcal{G}_{2k}$ of $\mathcal{G}$.
  - *Phase I*: Let $V_{2k+1} = \mathit{nobad}(\mathcal{G}_{2k})$. Then,
  $$f(\langle q, l\rangle) = 2k + 1 \ \text{ for all } \langle q, l\rangle \in V_{2k+1}.$$
  $$\mathcal{G}_{2k+1} = \mathcal{G}_{2k} \setminus V_{2k+1}.$$
  - *Phase II*: Let $V_{2k+2} = \mathit{finite}(\mathcal{G}_{2k+1})$. Then,
  $$f(\langle q, l\rangle) = 2k + 2 \ \text{ for all } \langle q, l\rangle \in V_{2k+2}.$$
  $$\mathcal{G}_{2k+2} = \mathcal{G}_{2k} \setminus V_{2k+2}.$$

  The graphs $\mathcal{G}_0$ as well all graphs $\mathcal{G}_{2k+2}$ are either empty, or guaranteed to have only infinite paths, since all nodes that can only access finitely many nodes are removed (a finite path ends in a node with no successor.)

Hence, if V_{2k} is non-empty Lemma 4.5.1 guarantees that V_{2k+1} is non-empty as well: there is a node in V_{2k+1} that accesses infinitely many vertices, but no B node. In particular there is an infinite path that is removed in *Phase I*. Hence, for some level l, all $l' > l$ satisfy that

$$width(\mathcal{G}_{2k+1}, l') + 1 \leq width(\mathcal{G}_{2k}, l')$$

Phase II only removes nodes, so

$$width(\mathcal{G}_{2k+2}, l') \leq width(\mathcal{G}_{2k+1}, l')$$

Since, initially $width(\mathcal{G}, l) \leq n$ for all levels l, it follows that, at the end of round k, for a sufficiently large l':

$$width(\mathcal{G}_{2k+1}, l') \leq n - (k+1)$$

Consequently, at the end of *Phase II* of round $n-1$: $width(\mathcal{G}_{2n-1}, l') \leq 0$. All remaining vertices in $\mathcal{G}_{2n-1}$ can access only finitely many vertices. Hence $\mathcal{G}_{2n} = \emptyset$, and the algorithm terminates. Note that it is possible that $\mathcal{G}_{2k} = \emptyset$ in an earlier round, but guaranteed that after round n, $\mathcal{G}_{2n} = \emptyset$.

It remains to be seen that f is indeed an odd S[1]-ranking.

The funcion f is a S[1]-ranking : By construction, all B vertices are marked in *Phase II* of some round because $B \cap nobad(\mathcal{G}_i) = \emptyset$, and hence receive an even value. Therefore, all B nodes satisfy condition (i) of the definition of S[1]-ranking. Now consider an arbitrary node $\langle q, l\rangle$. We consider three cases:

1. If $\langle q, l\rangle$ is removed in the Initial Stage then q is either a G node, in which case (ii) holds trivially, or it is in V_0. In the latter case, all its outgoing paths hit a G node in a finite number of steps, and all the intermediate nodes are mapped to 0. Hence, if $\langle q, l\rangle \rightarrow \langle q', l'\rangle$, then $f(\langle q', l'\rangle) = 0 = f(\langle q', l'\rangle)$ and $f(\langle q, l\rangle) \geq f(\langle q', l'\rangle)$, and condition (ii) holds.
2. If $\langle q, l\rangle$ is removed in *Phase I* of round k, then $\langle q, l\rangle \in V_{2k+1}$. Then all its outgoing paths either hit a node removed in a previous round or are in V_{2k+1}. In both cases $\langle q, l\rangle \rightarrow \langle q', l'\rangle$ implies $f(\langle q, l\rangle) \geq f(\langle q', l'\rangle)$.
3. If $\langle q, l\rangle$ is removed in *Phase II* of round k, then $\langle q, l\rangle \in V_{2k+2}$. Then all its outgoing paths either hit a node removed in a previous round, or are in V_{2k+1} or in V_{2k+2}. In all cases $\langle q, l\rangle \rightarrow \langle q', l'\rangle$ implies $f(\langle q, l\rangle) \geq f(\langle q', l'\rangle)$.

The function f is an odd S[1]-ranking : Consider an arbitrary path π. If π visits infinitely many G nodes, then the condition for f being odd on π holds. If π does not visit infinitely many G nodes, then f converges on π to some value. This value cannot be even, because that would imply that all these infinitely many vertices are in some V_{2k+2}, but there are not infinite paths containing these kind of node: by construction all nodes labeled in *Phase II* have finite outgoing paths before changing ranking.

This finishes the proof. □

4.5.2 An equivalent NBW

We describe here the translation from ASW[1] into NBW. The main idea is to encode in the states of the NBW cuts of a run DAG of the NBW, decorated with enough information to check whether an odd-ranking exists. In particular, each state of the alternating automaton present in a state of the NBW is labeled with a ranking value. This annotation must respect the definition of ranking in Def. 4.5.1. Additionally, the set of states of the ASW[1] that form a state of the NBW are partitioned into those that owe an improvement in the ranking (either a visit to a G state or a decrease in the ranking), and those

that already showed improvement. Membership to the owe set is propagated, so an accepting state is one in which all constituent states have seen some progress since the last accepting state. After an accepting state, the owe set is reset.

Formally, we start from an ASW[1] automaton $\mathcal{A} : \langle \Sigma, Q_{\mathcal{A}}, I_{\mathcal{A}}, \delta_{\mathcal{A}}, \{(B, G)\}\rangle$ and we build an NBW $N : \langle \Sigma, Q_N, I_N, \delta_N, F_N\rangle$ as follows:

- Q_N contains elements of the form (S, O, f) where $S \subseteq Q_{\mathcal{A}}$ is a subset of states of $\mathcal{A}$, $O \subseteq S$, and $f : S \to [2n]$ is a function that satisfies:

  Q1. If $q \in B$ then $f(q)$ is even.

- I_N contains all those $(M, O, f) \in Q_N$ where

  I1. M is a minimal model of $I_{\mathcal{A}}$ and $O = \{q \in M \mid q \notin G \text{ and } f(q) \text{ is even}\}$.

- $F_N = \{(S, O, f) \mid O = \emptyset\}$.

- $\delta_N : Q_N \times \Sigma \to 2^{Q_N}$, such that $(S', O', f') \in \delta_N((S, O, f), a)$ whenever there is one minimal model M_q of $\delta_{\mathcal{A}}(q, a)$ for each $q \in S$ satisfying:

  D1. $S' = \cup_{q \in S} M_q$,

  D2. For all $p \in S'$, the rank annotation $f'(p) \leq min\{f(q) \mid q \in pred(p) \setminus G\}$ where $pred(p) = \{q \in S \mid p \in M_q\}$ denotes the set of predecessors of p.

  D3. O' is given as follows. Let $p \in S' \setminus G$, we have
  - If $O = \emptyset$ then $p \in O'$ iff $f'(p)$ is even.
  - If $O \neq \emptyset$ then $p \in O'$ iff $f'(p) = f(q)$ for some $q \in (pred(p) \cap O)$.

The states of N consist of a set S representing elements of a cut of a run DAG of $\mathcal{A}$. The function f represents an S[1]-ranking, where **Q1** guarantees that no B node receives an odd value, and **D2** guarantees the non-increasing condition of rankings. Condition **D1** ensures that successor states of N correspond to legal successor cuts of a run of $\mathcal{A}$. Finally, condition **D3** ensures that O contains those vertices of the run DAG that have not seen progress for some path leading to them, where progress is defined as visiting a G state, or experiencing a decrease in f. A reset of this check is represented by a final state, which can happen only when all paths to all states contain some progress, as captured by F_N. Finally, **I1** captures that the initial states of N correspond to initial cuts of runs of $\mathcal{A}$. All these facts imply that a successful run DAG of $\mathcal{A}$ is matched by a successful run of N.

Theorem 4.5.3. *Let $\mathcal{A}$ be an* ASW[1] *and N the corresponding NBW. Then $w \in \mathcal{L}(\mathcal{A})$ if and only if $w \in \mathcal{L}(N)$.*

Proof. We prove the two directions separately:

"⇒" We assume $w \in \mathcal{L}(\mathcal{A})$ and show that $w \in \mathcal{L}(N)$. Let $\mathcal{G}$ be a run DAG for w on $\mathcal{A}$, and f an odd S[1]-ranking. Consider the sequence $Q_0 Q_1 \ldots$ of states of N induced by $\mathcal{G}$ and f as $Q_i = (S_i, O_i, f_i)$ with the set of states in S_i:

$$S_i \quad = \quad \{q \mid \langle q, i\rangle \in \mathcal{G}\}$$

and the pending states O_i:

$$\begin{array}{rcll} O_0 & = & \{p \mid \langle p, 0\rangle \in \mathcal{G} \text{ with } p \notin G \text{ and } f(\langle p, 0\rangle) \text{ is even}\} & \\ O_{i+1} & = & \{p \mid \langle p, i+1\rangle \in \mathcal{G} \text{ with } p \notin G \text{ and } f(\langle p, i+1\rangle) \text{ even}\} & \text{if } O_i = \emptyset \\ O_{i+1} & = & \{p \mid \langle p, i+1\rangle \in \mathcal{G} \text{ and } p \notin G, \text{ and for some } q \in O_i, & \\ & & \quad f(\langle p, i+1\rangle) = f(\langle q, i\rangle) \text{ and } \langle q, i\rangle \to \langle p, i+1\rangle\} & \text{if } O_i \neq \emptyset \end{array}$$

and

$$f_i(q) \quad = \quad f(\langle q, i\rangle)$$

It is routine to check that $Q_0Q_1\ldots$ is a run. We show that this run is accepting for N. By contradiction, if $Q_0Q_1\ldots$ is non accepting, there exists i such that, for all $j \geq i$, $Q_j \notin F_N$, hence $O_j \neq \emptyset$. By **D3** every $q_{j+1} \in O_{j+1}$ has a predecesor $q_j \in O_j$ with $f_j(q_j) = f_{j+1}(q_{j+1})$ being an even value by definition of O_i above. Since, as shown above, every $O_j \neq \emptyset$, it follows that there is an infinite sub-DAG of nodes in $\mathcal{G}$ of the form $\langle q_j, j\rangle$ with $f(\langle q_j, j\rangle)$ being even, and with infinitely many nodes having an incident edge. By König's lemma, since this DAG is finitely branching, it has an infinite path, all whose nodes are assigned the same even value by f. This is a contradiction with f being an odd S[1]-ranking for $\mathcal{G}$. Hence we find that $Q_0Q_1\ldots$ is a run that accepts w which shows $w \in \mathcal{L}(N)$.

"⇐" We assume now $w \in \mathcal{L}(N)$ and show that $w \in \mathcal{L}(\mathcal{A})$. Let $Q_0Q_1Q_2\ldots$ be an accepting run for w on N and let $\mathcal{G} = (V, E)$ and f be an induced run and function $V \to [2n]$. We conclude from **Q1** and **D2**, respectively, that properties (i) and (ii) of S[1]-ranking hold on f. Therefore f is an S[1]-ranking. Now let us show that f is an odd S[1]-ranking. To this end, consider an arbitrary path π in $\mathcal{G}$ for which we will show that either condition (i) or (ii) of the definition of odd S[1]-ranking holds. If π visits G nodes infinitely often, then condition (i) holds. Otherwise, there is an i after which no more G nodes are visited in π. Hence, since every node $\pi(i')$ with $i' > i+1$ has a predecessor not in G, **D2** shows that f converges on π to some value.

Let $j \geq i'$ be such that f has converged already (i.e., $f(\pi(j')) = f(\pi(j))$ for all $j' \geq j$). Let $Q_k, Q_l \in F_N$ with $j \leq k < l$ be two accepting states in the run Q_0Q_1 of w on N. It must be the case that $\pi(k+1) \notin O_{k+1}$. Assume the contrary (i.e. $\pi(k+1) \in O_{k+1}$), since π visits no G node after k and f has converged, then we conclude by **D3** that $\pi(k') \in O_{k'}$ for all $k' > k$, hence that $O_l \neq \emptyset$, and finally that $Q_l \notin F_N$ by definition of F_N which is a contradiction.

Also since $\pi(k+1) \notin O_{k+1}$ and $Q_k \in F_N$, **D3** shows that $f(\pi(k+1))$ is odd. Hence f converges on π to an odd value showing that condition (ii) of the definition of S[1]-ranking holds.

This concludes that f is an odd S[1]-ranking for $\mathcal{G}$. Finally Lem. 4.5.2 shows that $\mathcal{G}$ is an accepting run, hence that $w \in \mathcal{L}(\mathcal{A})$.

□

The automaton obtained with the previous construction can be easily pruned with one simple observation: if there is an odd S[1]-ranking, then there is an odd S[1]-ranking where all decrease (according to **D2**) only drop to the highest legal value. That is:

$$f'(p) = \begin{cases} M \text{ or } M-1 & \text{if } p \notin B \\ M \text{ or } M-2 & \text{if } p \in B \end{cases} \qquad \text{where } M = min\{f(q) \mid q \in pred(p) \setminus G\}$$

This observation reduces the guessing to 2 values, providing a more efficient ranking algorithm for all ASW[1] automata. The next section exploits the internal structure of stratified ASW[1] automata to introduce a faster solution, specific for the particular case of AHW.

4.5.3 Rankings for Stratified ASW[1]

Consider a stratified ASW[1] where Q is divided into strata $(S_1, \ldots, S_k)$ ordered according to $<$, and each stratum is labeled by α as either Büchi (all states are either B or G) or coBüchi (no state is G). The stratification structure implies that for every $q \in S_i$ and successor p with $p \in S_j$, either $S_j = S_i$ or $S_j < S_i$. This automata is equivalent to an AHW with

$$F = \bigcup_i \{S_i \cap G \mid \text{if } S_i \text{ is Büchi}\} \quad \cup \quad \bigcup_i \{S_i \cap B \mid \text{if } S_i \text{ is coBüchi}\}$$

and $H = \langle (S_1, \ldots, S_k), <, \alpha\rangle$. We use $m_j = |S_j|$ to refer to the number of states in stratum S_j. We first define the notion of stratified S[1]-ranking:

Definition 4.5.3. A stratified S[1]-ranking is a family of functions $f_j : S_j \to [2m_j]$ that satisfies:
(i) if $q \in S_j \cap B$ then $f_j(\langle q, l\rangle)$ is even,

(ii) for every $\langle q, l\rangle \to \langle q', l'\rangle$ in E with $q, q' \in S_j$, then $f_j(\langle q, l\rangle) \geq f_j(\langle q', l'\rangle)$, unless $q \in G$.

Intuitively, a stratified ASW[1] ranking is like an ASW[1] ranking except values need not decrease when moving across strata. Due to the stratification, every trace of a run gets trapped in a stratum of the automaton. Once the trace converges to a stratum, either the trace visits infinitely many good nodes, or the ranking converges to a single value. Again, the notion of odd ranking captures whether the suffix traces are accepting.

Definition 4.5.4 (odd stratified S[1]-ranking)**.** A stratified S[1]-ranking is odd whenever, for every infinite path π of $\mathcal{G}$, either

(i) π visits G states infinitely often, or

(ii) π gets trapped in stratum S_j and f_j converges to an odd value on π.

The following lemma justifies the construction of NBW using stratified rankings.

Lemma 4.5.4. *$\mathcal{G}$ is an accepting run iff there is a stratified odd* S[1]*-ranking for $\mathcal{G}$.*

Proof. We prove the two directions separately:

"$\Leftarrow$" Assume there is a strartified odd S[1]-ranking $\{f_j\}$ for $\mathcal{G}$ and let π be an arbitrary trace of $\mathcal{G}$. Since $\{f_j\}$ is odd, either π visits infinitely many G states, in which case π is accepting, or π converges to a stratum S_j and $f_j(\pi)$ converges to an odd value. In this second case, there is l such that for all $l' > l$, $f_j(\pi(l')) = f_j(\pi(l))$ and $f_j(\pi(l))$ is odd. By definition of stratified S[1]-ranking (point (i)), $\pi(l')$ cannot be a B state, and consequently π visits only finitely many B states. Hence, π is an accepting trace.

"$\Rightarrow$" Assume now that $\mathcal{G}$ is an accepting run for $\mathcal{A}$. The construction of each f_j works at each stratum independently. Fix S_j. First, one removes all G vertices and all those vertices not in S_j. The algorithm works exactly as with the proof of Lemma 4.5.2 by stages, at each stage first removing those states that cannot access B nodes, and then removeing those states that only access finitely many nodes. Since at each stage one removes at least one element from all cuts at a sufficiently large l, and the width of elements from S_j is at most $|S_j|$, the algorithm is guaranteed to finish in $|S_j|$ rounds, generating an odd S[1]-ranking for stratum S_j. It is routine to check that $\{f_j\}$ is indeed a stratified S[1]-ranking.

□

Stratified rankings drastically limit the guessing that is necessary in the construction of the states of the NBW, because each ranking is local to the stratum under consideration. The following choices produce a good ranking for stratum S_j, if there is such a good ranking:

G1. If S_j is accepting (no B states) then $f_j(q_j) = 1$ for all $q_j \in S_j$.

G2. If S_j is rejecting (only B states) then $f_j(q_j) = 2$ for all $q_j \in S_j$.

G3. If S_j is Büchi, then assign $f_j(q_j) = 2$ to $q_j \in B$, and $f_j(q_j) = 1$ to $q_j \in G$.

G4. If S_j is coBüchi then $f_j(q_j) \in [2m_j]$.

Note that this restriction eliminates the guessing except for coBüchi strata, and consequently ranking guessing only happens to the states of N_r in expressions $x\|r\rangle\!\rangle y$. In terms of the LTL fragment, all delays are one step so the size of $|N_r|$ is 1 and hence the maximum size of the coBüchi strata is 1. In fact, for LTL sub-expressions of the form $x\|r\rangle\!\rangle y$, N_r consists of a single B state, which can receive only value 2. Consequently, following the steps in this work, LTL expressions get translated into NBW of size $2^{O(n)}$.

4.5.4 An equivalent NBW using Stratified Rankings

We refine the construction for general ASW[1] rankings, limiting the guesses using **G1**-**G4**. Also, only predecessors within the same stratum are considered when computing f:

Q1s. If $q \in S_j \cap B$ then $f_j(q)$ is even.

D2s. $f'_j(p) \leq min\{f_j(q) \mid q \in pred(p) \setminus G\}$ where $pred(p) = \{q \in S_j \mid p \in M_q\}$ now only considers the same stratum.

D3s. O' is given as follows. Let $p \in S'_j \setminus G$, we have

- If $O = \emptyset$ then $p \in O'$ iff $f'_j(p)$ is even.
- If $O \neq \emptyset$ then $p \in O'$ iff $f'_j(p) = f_j(q)$ for some $q \in (pred(p) \cap O \cap S_j)$.

Theorem 4.5.5. *Let $\mathcal{A}$ be a stratified* ASW[1] *and N the corresponding NBW using stratified rankings. Then $w \in \mathcal{L}(\mathcal{A})$ if and only of $w \in \mathcal{L}(N)$.*

Proof. The proof is analogous to Theorem 4.5.3.

□

4.6 Empirical Evaluation

In this section we report the result of an empirical evaluation of the translation algorithms presented in this chapter. The evaluation was performed using Rounded, a sequential implementation written in OCaml, available at [SFS]. The running times reported in Figure 4.1 were run in an Intel Core2 @ 2.83GHz with 8GB of RAM running a 32 bit Linux kernel. Fig. 4.1 compares the number of states and the running time used to compute explicit NBW representations of two families of formulas (and their negation), for $i = 5, 8 \ldots 20$. These choices are inspired by [GO01]:

- $A_i = (p_1 \mathcal{U} (p_2 \mathcal{U} (\ldots \mathcal{U} p_i) \ldots)$. The expression A_i is equivalent to the RLTL expression $p_1|\mathbf{true}\rangle\!\rangle(p_2|\mathbf{true}\rangle\!\rangle \ldots)$.
- $B_i = p_1|\mathbf{true}^5\rangle\!\rangle(p_2|\mathbf{true}^5\rangle\!\rangle \ldots)$, where $\mathbf{true}^5$ stands for a five instant delay **true**;**true**;**true**;**true**;**true**. These are not expressible in LTL.

The table illustrates that the general ASW[1] ranking is only practical for the smallest cases. Limiting the guessing to the highest ranks allows to handle slightly larger examples. The stratified ranking translation results in a dramatic improvement, comparable to state of the art LTL translators, particularly considering that our prototype does not use simulations or handle propositional alphabets (only discrete alphabets). Simulation reductions have been reported [GO01] to be a very effective method to reduce the size of the NBW generated, but this optimization is currently ongoing work.

	APW		NBW (direct)		NBW (max2)		NBW (strat)	
	size	time(s)	size	time(s)	size	time(s)	size	time(s)
A_5	4	0.008	30	0.020	30	0.016	6	0.012
A_8	7	0.048	93	0.128	93	0.100	9	0.052
A_{11}	10	0.172	192	0.504	192	0.336	12	0.176
A_{14}	13	0.428	327	1.480	327	0.812	15	0.444
A_{17}	16	0.936	498	3.680	498	1.700	18	0.952
A_{20}	19	1.796	705	8.149	705	3.184	21	1.816
$\neg A_5$	4	0.008	46	0.032	34	0.020	6	0.012
$\neg A_8$	7	0.048	142	0.252	100	0.100	9	0.052
$\neg A_{11}$	10	0.172	292	1.140	202	0.336	12	0.172
$\neg A_{14}$	13	0.432	496	3.816	340	0.812	15	0.452
$\neg A_{17}$	16	0.940	754	10.545	514	1.688	18	0.948
$\neg A_{20}$	19	1.792	1066	25.718	724	3.160	21	1.884
B_5	20	0.048	782	2.900	782	0.780	22	0.068
B_8	35	0.268	2417	50.103	2417	5.936	37	0.296
B_{11}	50	1.084	4952	360.571	4952	26.846	52	0.952
B_{14}	65	2.316	8387	27m7s	8387	82.145	67	2.560
B_{17}	80	5.064	12722	1h56m	12722	217.698	82	5.200
B_{20}	95	10.041	17957	8h32m	17957	598.129	97	9.897
$\neg B_5$	20	0.048	1182	10.233	802	0.720	22	0.064
$\neg B_8$	35	0.268	3642	209.157	2452	5.704	37	0.300
$\neg B_{11}$	50	0.908	7452	26m56s	5002	25.106	52	0.956
$\neg B_{14}$	65	2.332	12612	3h13m	8452	81.129	67	2.404
$\neg B_{17}$	80	5.072	19122	18h15m	12802	220.762	82	5.192
$\neg B_{20}$	95	9.753	26982	58h8m	18052	674.070	97	9.905

Figure 4.1: Number of states (size) and the running time in seconds used to compute an APW$[0,1,2]$ and an NBW for some RLTL formulas.

5

Conclusion

The aim of this thesis was to develop new algorithms for the decidability and the model checking problems of RLTL. Our work includes two different approaches to this problems, in a first attempt to provide RLTL with decision procedures.

First, we have studied how bounded model checking techniques can be applied to RLTL and, as such, we introduced *bounded model checking for RLTL* . To this end, we have defined a new bounded semantics for RLTL that is suitable for finite segments of infinite words. We have also defined a bounded semantics based on three valued logic, which is more precise than the bi-valued bounded semantics allowing us to find counterexamples more quickly. Then, we have introduced a novel sound translation from RLTL expressions into SAT formulas for ultimately periodic words.

Second, we have presented a novel translation from the logic RLTL into alternating parity automata using only colors 0, 1 and 2, based on a bottom-up construction of specular pairs accepting complement languages. We have used game theory to prove that specular automata recognize complement languages. Inspired by the duality in the translation we introduced universal sequential operators that enriched the logic with negation normal forms. We have also shown that the resulting automata enjoy some stratified structure in their transition relation that makes all their strata purely Büchi or coBüchi. These automata are equivalent to hesitant automata. Then, we studied translations of the resulting automata into nondeterministic Büchi automata (NBW). The main result is the specialization of Street rankings to stratified automata to obtain a more efficient ranking translation. Unlike [KPV01] our construction preserves the alphabet between the alternating automaton an the NBW. To show the efficiency of our method, we have developed a prototype of the translation (available at [SFS]). The experimental results are very encouraging.

As part of our work in progress and potential future work, we are currently investigating alternative algorithms for model-checking RLTL specifications, e.g., based on antichains [WDMR08] and IC3 [Bra11]. We already have an advanced implementation of the antichains algorithm and we also plan to start the implementation of the bounded model checking for RLTL in short. All these efforts point to the development of a model checker for RLTL.

Bibliography

[AS04] Mohammad Awedh and Fabio Somenzi. Proving more properties with bounded model checking. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 96–108. Springer-Verlag, 2004.

[AS06] Mohammad Awedh and Fabio Somenzi. Termination criteria for bounded model checking: Extensions and comparison. *Electronic Notes in Theoretical Computer Science*, 144(1):51–66, 2006.

[BC03] Marco Benedetti and Alessandro Cimatti. Bounded model checking for past LTL. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 18–33. Springer-Verlag, 2003.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. In *Highly Dependable Software*, number 58 in Advances in Computers, chapter 3, pages 118–149. Academic Press, 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, LNCS, pages 193–207. Springer-Verlag, 1999.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BCP+06] Roderick Bloem, Alessandro Cimatti, Ingo Pill, Marco Roveri, and Simone Semprini. Symbolic implementation of alternating automata. In *11th International Conference on Implementation and Application of Automata (CIAA'06)*, volume 4094 of *LNCS*, pages 208–218. Springer-Verlag, 2006.

[BHJ+06] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5):1–64, 2006.

[Bor97] Arne Borälv. The industrial success of verification tools based on stålmarck's method. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 7–10, London, UK, 1997. Springer-Verlag.

[Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In *Proc. of VMCAI'11*, volume 6538 of *LNCS*, pages 70–87. Springer-Verlag, 2011.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[CKOS04] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 85–96. Springer-Verlag, 2004.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28:114–133, January 1981.

[DK08] Christian Dax and Felix Klaedtke. Alternation elimination by complementation. In *Proceedings of the 15th International Conference on Logic for Programming, Aritificial Intelligence and Reasoning (LPAR'08)*, volume 5530 of *LNCS*, pages 214–229. Springer-Verlag, 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, July 1960.

[FSW02] Alan M. Frisch, Daniel Sheridan, and Toby Walsh. A fixpoint based encoding for bounded model checking. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, volume 2517 of *LNCS*, pages 238–255. Springer-Verlag, 2002.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 53–65. Springer-Verlag, 2001.

[Har] Russel Harmer. Inocent game semantics. `http://www.pps.jussieu.fr/~russ/GS.pdf`. Course notes.

[HJK+06] Keijo Heljanko, Tommi A. Junttila, Misa KeinÃd'nen, Martin Lange, and Timo Latvala. Bounded model checking for weak alternating Büchi automata. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 95–108. Springer-Verlag, 2006.

[HJL05] Keijo Heljanko, Tommi A. Junttila, and Timo Latvala. Incremental and complete bounded model checking for full PLTL. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 98–111. Springer-Verlag, 2005.

[JJLR06] Markus Jehle, Jan Johannsen, Martin Lange, and Nicolas Rachinsky. Bounded model checking for all regular properties. *Electronic Notes in Theoretical Computer Science*, 144(1):3–18, 2006.

[JS07] Paul B. Jackson and Daniel Sheridan. A compact linear translation for bounded model checking. *Electronic Notes in Theoretical Computer Science*, 174(3):17–30, 2007.

[KPV01] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Extended temporal logic revisited. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'01)*, volume 2154 of *LNCS*, pages 519–535. Springer-Verlag, 2001.

[KV01] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic*, 2(3):408–429, 2001.

[KV04] Orna Kupferman and Moshe Y. Vardi. From complementation to certification. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 591–606. Springer-Verlag, 2004.

[KV05] Orna Kupferman and Moshe Y. Vardi. Complementation constructions for nondeterministic automata on infinite words. In *Proceedings of 11th International Conference for Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 206–221. Springer, apr 2005.

[Lan07] M. Lange. Linear time logics around PSL: Complexity, expressiveness, and a little bit of succinctness. In *CONCUR'07*, pages 90–104, 2007.

[LBHJ04] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi A. Junttila. Simple bounded LTL model checking. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 186–200. Springer-Verlag, 2004.

[LBHJ05] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi A. Junttila. Simple is better: Efficient bounded model checking for past LTL. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, LNCS, pages 380–395. Springer-Verlag, 2005.

[LS07] Martin Leucker and César Sánchez. Regular linear temporal logic. In Cliff Jones, Zhiming Liu, and Jim Woodcock, editors, *Proceedings of The 4th International Colloquium on Theoretical Aspects of Computing (ICTAC'07)*, volume 4711 of *LNCS*, pages 291–305, Macau, China, September 2007. Springer-Verlag.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag, 1995.

[MS87] David E. Muller and Paul E. Schupp. Altenating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–67, 1977.

[SC85] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear termporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.

[SFS] Julian Samborski-Forlese and César Sánchez. Rounded: An RLTL model checker. `http://software.imdea.org/rounded/`.

[SL10] César Sánchez and Martin Leucker. Regular linear temporal logic with past. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI'10)*, volume 5944 of *LNCS*, pages 295–311. Springer-Verlag, 2010.

[SS90] Gunnar Stålmarck and Martin Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety on Computer Control Systems (SAFECOMP'90)*, pages 31–36. Pergamon Press, Oxford, 1990.

[Sto74] Larry J. Stockmeyer. *The Computational Complexity of Word Problems*. PhD thesis, Massachusetts Institute of Technology, 1974.

[Str04] Ofer Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1):5–24, 2004.

[STV05] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: On hybrid approaches for LTL symbolic model checking. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 350–363. Springer-Verlag, 2005.

[Tho99] Wolfgang Thomas. Complementation of Büchi automata revisited. In *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 109–120. Springer-Verlag, 1999.

[WDMR08] Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-François Raskin. Antichains: Alternative algorithms for LTL satisfiability and model-checking. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4693 of *LNCS*, pages 63–77. Springer-Verlag, 2008.

[Zha97] Hantao Zhang. Sato: An efficient prepositional prover. In William McCune, editor, *Automated DeductionâĂŤCADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer Berlin / Heidelberg, 1997.