Regular Linear Temporal Logic*

Martin Leucker[1] and César Sánchez[2,3]

[1] Institut für Informatik
TU München, Germany
[2] Computer Science Department
Stanford University, Stanford, USA
[3] Computer Engineering Department
University of California, Santa Cruz, USA

Abstract. We present regular linear temporal logic (RLTL), a logic that generalizes linear temporal logic with the ability to use regular expressions arbitrarily as sub-expressions. Every LTL operator can be defined as a context in regular linear temporal logic. This implies that there is a (linear) translation from LTL to RLTL.
Unlike LTL, regular linear temporal logic can define all ω-regular languages, while still keeping the satisfiability problem in PSPACE. Unlike the extended temporal logics ETL$_*$, RLTL is defined with an algebraic signature. In contrast to the linear time μ-calculus, RLTL does not depend on fix-points in its syntax.

1 Introduction

We present *regular linear temporal logic* (RLTL), a formalism to express properties of infinite traces by conveniently *fusing* regular-expressions and linear-temporal logic. Moreover, we show that the satisfiability and equivalence of RLTL expressions are PSPACE-complete problems.

The linear temporal logic (LTL) [19, 16] is a modal logic over a linear frame, whose formulas express properties of infinite traces using two modalities: *next-time* and *until*. LTL is a widely accepted formalism for the specification and verification of concurrent and reactive systems. However, Wolper [26] showed that LTL cannot express all ω-regular properties (the properties expressible by finite-state automata on infinite words, known as Büchi automata [4]). In particular, it cannot express the property "p holds at every other moment". In spite of being a useful specification language, this lack of expressivity seems to surface in practice [20] and it has been pointed out (see for example [3]) that regular-expressions are sometimes very convenient in addition to LTL, in formal specifications. Actually, in the industry standard specification language PSL, arbitrary mixtures of regular expressions and LTL are allowed [1].

* Part of this work was done during the first author's stay at Stanford University and was supported by ARO DAAD190310197. The second author has been supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, and by NAVY/ONR contract N00014-03-1-0939.

To solve the expressivity problem, Wolper introduced the so called extended temporal logic ETL where new operators are defined as right linear grammars, and language composition is used to compose operators. ETL was later extended [25] to different kinds of automata. The main drawback of the extended temporal logics is that, in order to obtain the full expressivity, an infinite number of operators is needed.

An alternative approach consists on adapting the modal μ-calculus [5, 12] to the linear setting, which gives rise to the linear time μ-calculus, denoted as νTL [2]. Here, the full expressivity is obtained by allowing the use of fix point operators. It can be argued that this formalism is not algebraic either since one needs to specify recursive equations to describe temporal properties. Moreover, the only modality is the *nexttime*. Even though every ground regular expression can be translated into a νTL expression (see [14]), the concatenation operator cannot be directly represented in νTL, i.e., there is no context of νTL that captures concatenation. On the other hand, extending νTL with concatenation (the so-called fix point logic with chop FLC [18, 15]) allows expressing non-regular languages. This extra expressive power comes at the price of undecidable satisfiability and equivalence problems. A more restricted extension of νTL allowing only left concatenation with regular expressions is possible along the lines presented here, but this is out of the scope of this paper.

There have also been dynamic logics that try to merge regular expressions (for the program part) and LTL (for the action part), for example, Regular Process Logic [7]. However, it makes the satisfiability problem non-elementary by allowing arbitrary combinations of negations and regular operators. Dynamic linear-temporal logic DLTL [8] keeps the satisfiability problem in PSPACE, but restricts the use of regular expressions only as a generalization of the until operator. While the generalized until present in DLTL and the power operators present in RLTL are complementary (in the sense that none can be defined in terms of each other), the power operators are more suitable for extensions that can handle past, as discussed in Section 5.

An arbitrary mixture of (sequentially extended) regular expressions and LTL is possible in PSL [1, 6]. However, decision procedures for satisfiability etc. and their complexities are still an area of active research (for full PSL). Thus, RLTL can be understood as subset of PSL for which an efficient satisfiability procedure (PSPACE) is available.

The logic that we present here is a generalization of linear temporal logic and ω-regular expressions, based on the following observation. It is common for different formalisms to find the following three components in the (recursive) definition of operators:

1. *attempt*: an expression that captures the first try to satisfy the enclosing expression.
2. *obligation*: an expression that must be satisfied, if the attempt fails, to continue trying the enclosing expression. If both the attempt and the obligation fail, the sequence is not matched.

3. *delay*: an expression that describes when the enclosing expression must be started again.

For example, the binary Kleene-star z^*y matches a string s if either y (the attempt) matches s, or if after z (the delay), the whole expression z^*y matches the remaining suffix. In this case, no obligation is specified, so it is implicitly assumed to hold. Formally, the following equivalence holds $z^*y = y + z\,;z^*y$, or more explicitly

$$z^*y = y \;+\; (\Gamma^* \;\mid\; z\,;z^*y),$$

where $x \mid y$ denotes the intersection operator present in (semi-)extended regular expressions [22]. Consider also the linear temporal logic expression $x\,\mathcal{U}\,y$. An ω-sequence satisfies this expression if either y does (the attempt) or else, if x does (the obligation) and in the next step (the delay), the whole formula $x\,\mathcal{U}\,y$ holds. Formally,

$$x\,\mathcal{U}\,y = y \vee (x \wedge \bigcirc(x\,\mathcal{U}\,y)).$$

In Section 2 we will formalize this intuition by introducing a general operator that can be specialized for temporal logic and regular-expression constructs.

The rest of this document is structured as follows. Section 2 defines regular linear-temporal logic. Section 3 shows how to translate LTL and ω-regular expressions into RLTL. Section 4 shows, via a translation to alternating Büchi automata, that the logic defines only ω-regular languages, and that the satisfiability and equivalence problems are in PSPACE. Finally, Section 5 presents some concluding remarks.

2 Regular Linear Temporal Logic

We define in this section *regular linear temporal logic*, 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 languages over infinite words. Each formalism is defined as an algebraic signature, by giving meanings to the operators. We use Σ_{RE} for the operators in the language of regular expressions, Σ_{TL} for the signature of the language for infinite words, and Σ as a short hand for $\Sigma_{\text{RE}} \cup \Sigma_{\text{TL}}$.

We begin by fixing a finite set of propositions *Prop*, and from it the alphabet $\Gamma = 2^{Prop}$ of input actions (observable properties of individual states). As usual, Γ^* denotes the finite sequences of words over Γ, Γ^ω stands for the set of infinite words, and Γ^∞ is $\Gamma^* \cup \Gamma^\omega$. Given a word w, we use $pos(w)$ to denote the set of positions of w: if $w \in \Gamma^\omega$ then $pos(w)$ is $\{1, \ldots\} = \omega$; if $w \in \Gamma^*$ then $pos(w) = \{1, \ldots, |w|\}$, where $|w|$ denotes the length of w as usual. We use $w[i]$ to denote the letter from Γ at position i of w. We use *Pos* to denote the set of positions of words in Γ^ω, i.e., *Pos* is an alias of ω.

2.1 Regular Expressions

We first introduce a variation of regular expressions that can define regular languages that do not contain the empty word. Basic expressions are boolean

combinations of elements from $\mathbb{B}(\mathit{Prop})$ that identify the elements of Γ, including **true** for *Prop* and **false** for $\varnothing$.

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

$$\alpha ::= \alpha + \alpha \quad \Big| \quad \alpha \,;\, \alpha \quad \Big| \quad \alpha^* \alpha \quad \Big| \quad p \tag{1}$$

where p ranges over basic expressions. The operators $+$, ; and $*$ define the standard union, concatenation and binary Kleene-star[4]. The signature of regular expressions is then

$$\Sigma_{\text{RE}} = \{\mathbb{B}(\mathit{Prop})^0,\ +^2,\ ;^2,\ *^2\}$$

where the superindices indicate the arity of the operators. The set of regular expressions RE is the set of all ground expressions over this signature. Note that this signature contains no variables or fix-point quantifiers.

Semantics To ease the definition of RLTL for infinite languages, we define regular expressions as accepting *segments* of an infinite word. Given an infinite word w and two positions i and j, the tuple (w, i, j) is called a segment of the word w. Similarly, (w, i) is called a *pointed word*. The semantics of regular expressions is described by defining a relation $\vDash_{\text{RE}}$ that relates expressions with their sets of segments, that is $\vDash_{\text{RE}} \subseteq (\Gamma^\omega \times \mathit{Pos} \times \mathit{Pos}) \times \text{RE}$. The semantics is defined inductively as follows. Given a proposition $p \in \mathit{Prop}$, expressions x, y, and z, and a word w,

- $(w, i, j) \vDash_{\text{RE}} p$ whenever $w[i]$ satisfies p and $j = i + 1$.
- $(w, i, j) \vDash_{\text{RE}} x + y$ whenever either $(w, i, j) \vDash_{\text{RE}} x$ or $(w, i, j) \vDash_{\text{RE}} y$, or both.
- $(w, i, j) \vDash_{\text{RE}} x \,;\, y$ whenever for some $k \in \mathit{pos}(w)$, $(w, i, k) \vDash_{\text{RE}} x$ and $(w, k, j) \vDash_{\text{RE}} y$.
- $(w, i, j) \vDash_{\text{RE}} x^* y$ whenever either $(w, i, j) \vDash_{\text{RE}} y$, or for some sequence $(i_0 = i, i_1, \ldots i_m)$ $\quad (w, i_k, i_{k+1}) \vDash_{\text{RE}} x$ and $(w, i_m, j) \vDash_{\text{RE}} y$.

The semantical style used above, more conventional for temporal logics, is equivalent to the more classical of associating a language over finite words to a given expression: for $v \in \Gamma^*$, $v \in \mathcal{L}(x)$ whenever for some $w \in \Gamma^\omega$, $(vw, 1, |v|) \vDash_{\text{RE}} x$. In this manner the definition of $*$ is equivalent to the conventional definition, that is, both describe the same language:

$$\mathcal{L}(x^* y) = \mathcal{L}(\sum_{i \geq 0} x^i \,;\, y)$$

where $x^i \,;\, y$ is defined inductively as $x^0; y = y$ and $x^{i+1} = x \,;\, x^i$, as usual. Since p satisfies that if $(w, i, j) \vDash_{\text{RE}} p$ then $j > i$, it follows the empty word is not in

[4] Stephen C. Kleene himself in [11] introduced the $*$ operator as a binary operator. Our choice of a binary $*$ is determined by our key decision of defining languages that do not contain the empty word. An alternative is to introduce a unary x^+ operator.

$\mathcal{L}(p)$, and also that $\mathcal{L}(x + y)$ and $\mathcal{L}(x\ ;\ y)$ cannot contain the empty word. It also follows that x^*y cannot contain the empty word: if v is in $\mathcal{L}(z^*y)$ then v is in $\mathcal{L}(z^k y)$ for some k.

Moreover, every regular language over finite words (that does not contain the empty word) can be defined, since x^+ is equivalent to x^*x.

2.2 Regular Linear Temporal Logic over Infinite Words

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.

Syntax The set of RLTL expressions is the smallest set closed under:

$$\phi ::= \phi \vee \phi \ \mid\ \phi \wedge \phi \ \mid\ \alpha\ ;\ \phi \ \mid\ \alpha^{\phi}\phi \ \mid\ \alpha_{\phi}\phi \ \mid\ \widehat{\alpha} \tag{2}$$

where α ranges over regular expressions RE. The symbols $\vee$ and $\wedge$ stand for the conventional union and intersection of languages (i.e., conjunction and disjunction in logics and $+$ 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 $\alpha^{\phi}\phi$, called the power operator, and its dual $\alpha_{\phi}\phi$ allow simple recursive definitions, including the Kleene-star (x^{ω} for infinite words) and the various operators in linear temporal logic. Finally, $\widehat{\alpha}$ denotes the suffix closure (arbitrary extension of a set of finite words to infinite words). The signature of RLTL is then:

$$\Sigma_{\text{TL}} = \{\vee^2,\ \wedge^2,\ \ ;^2,\ (\cdot^{\cdot}\cdot)^3\ ,\ \ (\cdot_{\cdot}\cdot)^3,\ \widehat{\cdot}^{1}\}$$

where the superindices again indicate the arity of the operators. Even though the symbol ; is overloaded we consider the signatures to be disjoined. The operators $\vee$ and $\wedge$ require two expressions in the language of Σ_{TL}, while $(\cdot^{\cdot}\cdot)$, $(\cdot_{\cdot}\cdot)$ and $\widehat{\cdot}$ require the first argument to be an expression in the language of Σ_{RE} and the rest in Σ_{TL}. The set of regular linear temporal logic expressions RLTL is the set of all ground expressions over this signature. Note again that this signature contains no variable or fix-point quantifier.

Semantics The semantics of an RLTL expression is defined as a binary relation $\vDash$ between pointed words and expressions, that is $\vDash \subseteq (\Gamma^{\omega} \times Pos) \times RLTL$. This relation is defined inductively as follows. Given RLTL expressions x and y and regular expression z:

- $(w, i) \vDash x \vee y$ whenever either $(w, i) \vDash x$ or $(w, i) \vDash y$, or both.
- $(w, i) \vDash x \wedge y$ whenever both $(w, i) \vDash x$ and $(w, i) \vDash y$.
- $(w, i) \vDash z\ ;\ y$ whenever for some $k \in pos(w)$, $(w, i, k) \vDash_{\text{RE}} z$ and $(w, k) \vDash y$.
- $(w, i) \vDash z^x y$ whenever $(w, i) \vDash y$ or for some sequence $(i_0 = i, i_1, \ldots i_m)$ $(w, i_k, i_{k+1}) \vDash_{\text{RE}} z$ and $(w, i_k) \vDash x$, and $(w, i_m) \vDash y$.

– $(w,i) \vDash z_x y$ whenever one of:
  (i) $(w,i) \vDash y$ and $(w,i) \vDash x$
  (ii) for some sequence $(i_0 = i, i_1, \ldots i_m)$
    $(w, i_k, i_{k+1}) \vDash_{\text{RE}} z$ and $(w, i_k) \vDash y$ and $(w, i_m) \vDash x$
  (iii) for some infinite sequence $(i_0 = i, i_1, \ldots)$
    $(w, i_k, i_{k+1}) \vDash_{\text{RE}} z$ and $(w, i_k) \vDash y$
– $(w,i) \vDash \hat{z}$ whenever for some $k \in pos(w)$, $(w,i,k) \vDash_{\text{RE}} z$.

The semantics of $z^x y$ establish that either the obligation y is satisfied at the point i of evaluation, or there is a sequence of delays—as determined by z—after which y holds, and x holds after each individual delay. The semantics of $z_x y$ establish that y must hold initially and after each delay—as determined by z— and that x determines when the repetition of the delay can stop (if it stops at all).

As with regular expressions, languages can also be associated with RLTL expressions in the standard form: a word $w \in \Gamma^\omega$ is in the language of an expression x, denoted by $w \in \mathcal{L}(x)$, whenever $(w,1) \vDash x$. The following lemmas hold immediately from the definitions:

Lemma 1. *For every RLTL expressions x and y and* RE *expression z:*

– *The expression $z^x y$ is equivalent to $y \vee (x \wedge z\,;\, z^x y)$.*
– *The expression $z_x y$ is equivalent to $y \wedge (x \vee z\,;\, z_x y)$.*

Lemma 2. *If L_x is the language of x, L_y is the language of y and L_z the language of z, then*

– *The language of $z^x y$ is the least fix-point solution of the equation:*

$$X = L_y \cup (L_x \cap L_z; X)$$

– *The language of $z_x y$ is the greatest fix-point solution of the equation:*

$$X = L_y \cap (L_x \cup L_z; X)$$

where ; is the standard language concatenation.

Thus, although the semantics of the power operators is not defined using fix point equations, it can be characterized by such equations, similar as the until operator in LTL.

We finish this section by justifying the need of the operator $\widehat{\alpha}$ in RLTL. It is clear, directly from the semantics, that the operators $\wedge$, $\vee$ and ; will not define infinite languages (or equivalently pointed models) unless their arguments do. By Lemma 1, the same holds for the power and dual power operators. The expression $\widehat{x}$ serves as a *pump* of the finite models (segments) of x to any continuation. An alternative would have been to include a universal expression ($\top$ in the next section) from which $\widehat{x} = x\,;\top$. Similarly, $\top = \widehat{true}$, so both alternatives are equivalent.

In the sequel, the *size* of an RLTL formula is defined as the total number of its symbols.

3 Translating LTL and Regular Expressions into RLTL

We will use $\top$ and $\bot$ as syntactic sugar for $\widehat{\mathbf{true}}$ and $\widehat{\mathbf{false}}$ (resp). In particular, observe that $(w,i) \models \top$ and $(w,i) \not\models \bot$ for every pointed word (w,i). We first introduce some equivalences of RLTL, very simple to prove, that will assist in our definitions:

$$\begin{array}{ll} x \vee y = y \vee x & x \wedge y = y \wedge x \\ \top \vee x = \top & \top \wedge x = x \\ \bot \vee x = x & \bot \wedge x = \bot \end{array}$$

3.1 Translating ω-regular expressions

First, we show how to translate ω-regular expressions into regular linear temporal logic. An ω-regular expression is of the form:

$$\sum_i x_i \,;\, (y_i)^\omega$$

for a finite family of regular expressions x_i and y_i. Note that when a more conventional definition of regular expressions is used (one that allows the definition of languages containing the empty word), one must explicitly require that y_i does not posses the empty word property, which is not needed in our definition. Also, the case of x_i possessing the empty word property (in the classical definition), can be handled easily since for every y_i the following equivalence holds: $y_i^\omega = y_i \,;\, (y_i^\omega)$. Then every expression of the form $x_i \,;\, (y_i)^\omega$ can be translated into $(x_i \,;\, y_i) \,;\, (y_i^\omega)$, for which the finite prefix does not accept the empty word and it is in the variation of regular expressions introduced here.

Lemma 3. *Given a regular expression z, the regular linear temporal logic expression $z_\bot\top$ is equivalent to z^ω.*

Proof. As no pointed word (w,i) satisfies $\bot$, the only relevant case in the semantics of the dual power operator for $z_\bot\top$ is that there is an infinite sequence of points $(i_1, i_2, \ldots)$ for which $(w, i_k, i_{k+1}) \models z$. Therefore $w \in \mathcal{L}(z^\omega)$. □

It follows that the ω-regular expression $\sum_i x_i ; (y_i)^\omega$ is equivalent to the RLTL expression $\bigvee_i x_i \,;\, (y_{i\,\bot}\top)$. This immediately implies:

Corollary 1. *The following are true for regular linear temporal logic:*
- *RLTL can express every ω-regular language.*
- *The set of operators $\{\vee, ;,$ dual power$\}$ is complete.*

Observe that no alternation of the power operators is needed to obtain expressive completeness (or in terms of Lemma 2 no alternation of fix points is necessary). This result is analogous to the linear μ-calculus [14], where the alternation hierarchy collapses at level 0 (in terms of expressiveness).

3.2 Translating LTL

We consider the following definition of LTL:

$$\psi ::= p \;\mid\; \psi \vee \psi \;\mid\; \psi \wedge \psi \;\mid\; \bigcirc\psi \;\mid\; \Box\psi \;\mid\; \Diamond\psi \;\mid\; \psi\,\mathcal{U}\,\psi \;\mid\; \psi\,\mathcal{R}\,\psi$$

which allows to express every linear temporal logic property in negation normal form. Note that $\Box\psi$ and $\Diamond\psi$ are just added for convenience.

The semantics of LTL expressions are defined, similarly to RLTL, by defining a binary relation $\vDash_{\mathrm{LTL}}$ between pointed words and LTL expressions: $\vDash_{\mathrm{LTL}} \subseteq (\Gamma^\omega \times \mathit{Pos}) \times \mathit{LTL}$. The semantics is defined inductively. The basic expressions and boolean operators are mapped as conventionally. Let x and y be arbitrary LTL expressions. The semantics of the temporal operators is:

- $(w,i) \vDash_{\mathrm{LTL}} \Diamond x$ whenever $(w,j) \vDash_{\mathrm{LTL}} x$ for some $j \geq i$.
- $(w,i) \vDash_{\mathrm{LTL}} \Box x$ whenever $(w,j) \vDash_{\mathrm{LTL}} x$ for all $j \geq i$.
- $(w,i) \vDash_{\mathrm{LTL}} \bigcirc x$ whenever $(w,i+1) \vDash_{\mathrm{LTL}} x$.
- $(w,i) \vDash_{\mathrm{LTL}} x\,\mathcal{U}\,y$ whenever $(w,j) \vDash_{\mathrm{LTL}} y$ for some $j \geq i$, and $(w,k) \vDash_{\mathrm{LTL}} x$ for all $i \leq k < j$.
- $(w,i) \vDash_{\mathrm{LTL}} x\,\mathcal{R}\,y$ whenever $(w,j) \vDash_{\mathrm{LTL}} y$ for all $j \geq i$, or for some j, $(w,j) \vDash_{\mathrm{LTL}} x$ and for all k within $i \leq k < j$, $(w,j) \vDash_{\mathrm{LTL}} y$.

Consider the following procedure, that translates an LTL expression ψ into an RLTL expression $\tau(\psi)$:

- $\tau(p) = \widehat{p}$, $\tau(x \wedge y) = \tau(x) \wedge \tau(y)$, $\tau(x \vee y) = \tau(x) \vee \tau(y)$,
- $\tau(\bigcirc x) = \mathbf{true}\,;\tau(x)$,
- $\tau(\Box x) = \mathbf{true}_{\perp}\tau(x)$,
- $\tau(\Diamond x) = \mathbf{true}^{\top}\tau(x)$,
- $\tau(x\,\mathcal{U}\,y) = \mathbf{true}^{\tau(x)}\tau(y)$,
- $\tau(x\,\mathcal{R}\,y) = \mathbf{true}_{\tau(x)}\tau(y)$.

Theorem 1. *Every LTL expression defines the same language as its RLTL translation.*

Proof. The proof proceeds by structural induction. For the basic expression, the boolean operators and $\bigcirc$ the result holds directly from the definitions. We show here the equivalence for $\mathcal{U}$ (the rest follow similarly). It is well known that $x\,\mathcal{U}\,y$ is the least fix point solution of the equation $X \equiv y \vee (x \wedge \bigcirc(x\,\mathcal{U}\,y))$, which is by Lemma 2 the semantics of $\mathbf{true}^{\tau(x)}\tau(y)$. □

Our translation maps every LTL operator into an equivalent RLTL context (with the same number of *holes*). Consequently, this translation only involves a linear blow-up in the size of the original formula. Since checking satisfiability of linear temporal logic is PSPACE-hard [21] this translation immediately gives a lower bound on the complexity of RLTL.

Proposition 1. *The problems of satisfiability and equivalence for regular linear temporal logic are PSPACE-hard.*

4 Translating RLTL into Alternating Automata

We now show that every RLTL formula can be translated with a linear blow-up into an alternating automaton accepting precisely its models.

4.1 Preliminaries

Let us, however, first recall the definitions of (nondeterministic) automata operating on finite words and alternating Büchi automata operating on infinite words.

A *nondeterministic finite automaton* (NFA) is a tuple $\mathcal{A} : \langle \Gamma, Q, q_0, \partial, F \rangle$ where Γ is the alphabet, Q a finite set of *states*, $q_0 \in Q$ the *initial state*, $\partial : Q \times \Gamma \to 2^Q$ the *transition function*, and $F \subseteq Q$ is the set of *final states*. An NFA operates on finite words: A *run* of $\mathcal{A}$ on a word $w = a_1 \dots a_n \in \Gamma^*$ is a sequence of states and actions $\rho = q_0 a_1 q_1 \dots q_n$, where q_0 is the initial state of $\mathcal{A}$ and for all $i \in \{1, \dots n\}$, we have $q_{i+1} \in \partial(q_i, a_i)$. The run is called *accepting* if $q_n \in F$. The *language* of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of words $w \in \Gamma^*$ for which an accepting run exists.

For a finite set $\mathcal{X}$ of variables, let $\mathcal{B}^+(\mathcal{X})$ be the set of *positive Boolean formulas* over $\mathcal{X}$, i.e., the smallest set such that $\mathcal{X} \subseteq \mathcal{B}^+(\mathcal{X})$, $\mathbf{true}, \mathbf{false} \in \mathcal{B}^+(\mathcal{X})$, and $\phi, \psi \in \mathcal{B}^+(\mathcal{X})$ implies $\phi \wedge \psi \in \mathcal{B}^+(\mathcal{X})$ and $\phi \vee \psi \in \mathcal{B}^+(\mathcal{X})$. We say that a set $Y \subseteq \mathcal{X}$ *satisfies* (or is a *model* of) a formula $\phi \in \mathcal{B}^+(\mathcal{X})$ iff ϕ evaluates to **true** when the variables in Y are assigned to **true** and the members of $\mathcal{X} \backslash Y$ are assigned to **false**. A model is called *minimal* if none of its proper subsets is a model. For example, $\{q_1, q_3\}$ as well as $\{q_2, q_3\}$ are minimal models of the formula $(q_1 \vee q_2) \wedge q_3$.

An *alternating Büchi automaton* (ABA) is a tuple $\mathcal{A} : \langle \Gamma, Q, q_0, \partial, F \rangle$ where Γ, Q, and F are as for NFAs. The transition function ∂, however, yields a positive boolean combination of successor states: $\partial : Q \times \Gamma \to \mathcal{B}^+(Q)$. Furthermore, an ABA operates on infinite words: A *run* over an infinite word $w = a_0 a_1 \dots \in \Gamma^\omega$ is a Q-labeled directed acyclic graph (V, E) such that there exist labellings $l : V \to Q$ and $h : V \to \mathbb{N}$ which satisfy the following properties:

- there is a single $v_0 \in V$ with $h(v_0) = 0$. Moreover, $l(v_0) = q_0$.
- for every $(v, v') \in E$, $h(v') = h(v) + 1$.
- for every $v' \in V$ with $h(v') \geq 1$, $\{v \in V \mid (v, v') \in E\} \neq \varnothing$,
- for every $v, v' \in V$, $v \neq v'$, $l(v) = l(v')$ implies $h(v) \neq h(v')$, and
- for every $v \in V$, $\{l(v') \mid (v, v') \in E\}$ is a minimal model of $\partial(l(v), a_{h(v)})$.

A run (V, E) is *accepting* if every maximal finite path ends in a node $v \in V$ with $\partial(l(v), a_{h(v)}) = \mathbf{true}$ and every maximal infinite path, wrt. the labeling l, visits at least one final state infinitely often. The language $\mathcal{L}(\mathcal{A})$ of an automaton $\mathcal{A}$ is determined by all strings for which an accepting run of $\mathcal{A}$ exists. We also consider *alternating co-Büchi automaton* (AcBA), defined exactly as ABA, except that the accepting condition establishes that all final states are visited only finitely many times in accepting paths.

We measure the *size* of an NFA, ABA and AcBA in terms of its number of states.

An ABA is *weak* (WABA), if there exists a partition of Q into disjoints sets Q_i, such that for each set Q_i either $Q_i \subseteq F$ or $Q_i \cap F = \emptyset$, and, there is a partial order $\leq$ on the collection of the Q_i's such that for every $q \in Q_i$ and $q' \in Q_j$ for which q' occurs in $\delta(q, a)$, for some $a \in \Gamma$, we have $Q_j \leq Q_i$.

It was shown in [13] that every AcBA can be translated into a WABA with a quadratic blow-up. Furthermore, it was shown in [17] that for an ABA accepting L, we get an AcBA accepting the complement of L, when dualizing the transition function (switching $\wedge$ with $\vee$ and **true** with **false**) and turning the acceptance condition into a co-Büchi acceptance condition. This gives

Proposition 2. *For every ABA $\mathcal{A}$ with n states, there is an ABA $\bar{\mathcal{A}}$ with at most n^2 states accepting the complement of $\mathcal{A}$'s language.*

4.2 Translation

We are now ready to formulate the main theorem of this section:

Theorem 2. *For every $\phi \in RLTL$, there is an ABA $\mathcal{A}_\phi$ accepting precisely the ω-words satisfying ϕ. Moreover, the size of $\mathcal{A}_\phi$ is linear in the size of ϕ.*

Corollary 2. *Checking satisfiability of an RLTL formula is PSPACE-complete.*

Proof. By Proposition 1, satisfiability of an RLTL formula is PSPACE-hard. Given $\phi \in RLTL$, we can construct $\mathcal{A}_\phi$ according to Theorem 2, and check $\mathcal{A}_\phi$ for emptiness, which can be done in PSPACE [24].

As usual, we call two formulas of $RLTL$ *equivalent* iff their sets of models coincide.

Lemma 4. *Checking equivalence of two RLTL formulas is PSPACE-complete.*

Proof. By Proposition 1, equivalence of two RLTL formulas ϕ and ψ is PSPACE-hard.

The formulas ϕ and ψ are equivalent iff both $(\neg\phi \wedge \psi)$ and $(\phi \wedge \neg\psi)$ are unsatisfiable. Even though complementation is not present in RLTL, we can use automata constructions to perform these two tests. The construction of Theorem 2 gives ABA $\mathcal{A}_\phi$ and $\mathcal{A}_\psi$ polynomial in the size of the formula. By Proposition 2, we can complement an ABA with an at most quadratic blow-up. ABAs in turn can be combined with $\wedge$. The check for emptiness of the resulting alternating automata can be done in PSPACE [24]. □

In the remainder of this section, we present the construction of $\mathcal{A}_\phi$ for $\phi \in RLTL$, hereby proving Theorem 2. The procedure works bottom-up the parse tree of ϕ. Recall that every regular expression α can be translated into an equivalent NFA [9].

Now, consider alternating Büchi automata for x and y, $\mathcal{A}_x : \langle \Gamma, Q^x, q_o^x, \partial^x, F^x \rangle$ and $\mathcal{A}_y : \langle \Gamma, Q^y, q_o^y, \partial^y, F^y \rangle$, and a non-deterministic automaton (over finite words) for z: $\mathcal{A}_z : \langle \Gamma, Q^z, q_o^z, \partial^z, F^z \rangle$. Without loss of generality, we assume that their state spaces are disjoint. The construction is sketched visually in the appendix. We consider the different operators of RLTL:

Disjunction The automaton for $x \vee y$ is:

$$\mathcal{A}_{x \vee y} \ : \ \langle \Gamma, Q^x \cup Q^y, q_0, \partial, F^x \cup F^y \rangle$$

where q_0 is a fresh new state. The transition function is defined as

$$\partial(q,a) = \begin{cases} \partial^x(q,a) & \text{if } q \in Q^x \\ \partial^y(q,a) & \text{if } q \in Q^y \end{cases}$$
$$\partial(q_0,a) = \partial^x(q_0^x,a) \vee \partial^y(q_0^y,a).$$

Thus, from the fresh initial state q_0, $\mathcal{A}_{x \vee y}$ chooses non-deterministically one of the successor states of $\mathcal{A}_x$'s or $\mathcal{A}_y$'s initial state. Clearly, the accepted language is the union.

Conjunction The automaton for $x \wedge y$ is:

$$\mathcal{A}_{x \wedge y} \ : \ \langle \Gamma, Q^x \cup Q^y, q_0, \partial, F^x \cup F^y \rangle$$

where q_0 is again a fresh new state. The transition function is defined as before except

$$\partial(q_0,a) = \partial^x(q_0^x,a) \wedge \partial^y(q_0^y,a).$$

Hence, from the fresh initial state q_0, $\mathcal{A}_{x \wedge y}$ follows both $\mathcal{A}_x$'s and $\mathcal{A}_y$'s initial state. Clearly, the accepted language is the intersection.

Suffix extensions The automaton for $\widehat{z}$ is:

$$\mathcal{A}_{\widehat{z}} \ : \ \langle \Gamma, Q^z \cup \{q_{tt}\}, q_0^z, \partial, \{q_{tt}\} \rangle$$

where q_{tt} is a fresh new state and ∂ is defined, for $q \in Q^z$ as:

$$\partial(q,a) = \begin{cases} \bigvee\{\partial^z(q,a)\} & \text{if } q \notin F^z \\ \bigvee\{\partial^z(q,a)\} \vee \{q_{tt}\} & \text{if } q \in F^z \end{cases}$$

and $\partial(q_{tt},a) = q_{tt}$. Thus, from a final state, which signals that the prefix of the infinite word read so far matches the regular expression, the automaton may non-deterministically choose to accept the remainder of the word.

Concatenation The automaton for $z\,;x$ is:

$$\mathcal{A}_{z;x} \ : \ \langle \Gamma, Q^z \cup Q^x, q_0^z, \partial, F^x \rangle$$

where ∂ is defined, for $q \in Q^z$ as:

$$\partial(q,a) = \begin{cases} \bigvee\{\partial^z(q,a)\} & \text{if } \partial^z(q,a) \cap F^z = \emptyset \\ \bigvee\{\partial^z(q,a)\} \vee q_0^x & \text{if } \partial^z(q,a) \cap F^z \neq \emptyset \end{cases}$$

and, for $q \in Q^x$ as $\partial(q,a) = \partial^x(q,a)$. Recall that $\mathcal{A}_z$ is a nondeterministic automaton. Whenever $\mathcal{A}_z$ can non-deterministically choose a successor that is a final state, it can also switch to $\mathcal{A}_x$. Thus, the accepted language is indeed the concatenation.

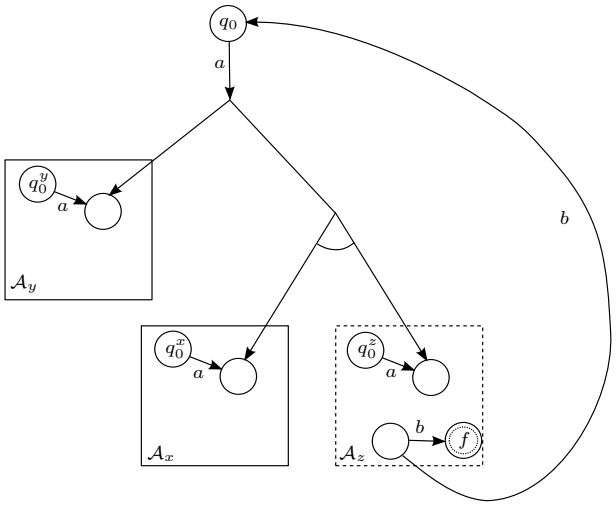

Fig. 1. Construction for the power operator

Power The automaton for $z^x y$ is:

$$\mathcal{A}_{z^x y} \ : \ \langle \Gamma, Q^z \cup Q^x \cup Q^y \cup \{q_0\}, q_0, \partial, F^x \cup F^y \rangle$$

where ∂ is defined as follows. The successor for a of the initial state is:

$$\partial(q_0, a) = \partial^y(q_0^y, a) \vee (\partial^x(q_0^x, a) \wedge \bigvee\{\partial^z(q_0^z, a)\})$$

The successor of Q^x and Q^y are defined as in $\mathcal{A}_x$ and $\mathcal{A}_y$, i.e., $\partial^x(q, a)$ for $q \in Q^x$, $\partial^y(q, a)$ for $q \in Q^y$. For $q \in Q_z$

$$\partial(q, a) = \begin{cases} \bigvee\{\partial^z(q, a)\} & \text{if } \partial^z(q, a) \cap F^z = \emptyset \\ \bigvee\{\partial^z(q, a)\} \vee q_0 & \text{if } \partial^z(q, a) \cap F^z \neq \emptyset \end{cases}$$

The construction, depicted in Fig. 1, follows precisely the equivalence $z^x y \equiv y \vee (x \wedge z; z^x y)$ established in Lemma 1 and the construction for disjunction, conjunction, and concatenation.

Dual power The automaton for $z_x y$ is:

$$\mathcal{A}_{z^x y} \ : \ \langle \Gamma, Q^z \cup Q^x \cup Q^y \cup \{q_0\}, q_0, \partial, F^x \cup F^y \cup \{q_0\} \rangle$$

where ∂ is defined exactly as before except for the successor for a of the initial state:

$$\partial(q_0, a) = \partial^y(q_0^y, a) \wedge (\partial^x(q_0^x, a) \vee \bigvee\{\partial^z(q_0^z, a)\})$$

Note, however, the state q_0 is now accepting, since the evaluation is allowed to loop in z for ever, restarting a copy of y at each repetition of z.

Complexity Recall that a regular expression can linearly be translated into a corresponding NFA [9, 10]. Examining the construction given above, we see that each operator adds at most one extra state. Thus, the overall number of states of the resulting automaton is linear with respect to the size of the formula.

The above construction for the concatenation operator relies heavily on the fact that the automaton $\mathcal{A}_z$ for a regular expression is nondeterministic. If RLTL were based on extended regular expressions, which offer boolean combinations of regular expressions (including negation), there would be no hope to get a PSPACE satisfiability procedure, as checking emptiness for extended regular expressions is already of non-elementary complexity [22]. On the same line, semi-extended regular expressions (that add *conjunction* to regular expressions) and input-synchronizing automata as introduced by Yamamoto [27] do neither give a PSPACE algorithm.

5 Conclusion and Discussion

In this paper, we introduced RLTL, a temporal logic that allows to express all ω-regular properties. It allows a smooth combination of LTL-formulas and regular expressions. Besides positive boolean combinations, only two *power* operators are introduced, which generalize LTL's *until* as well as the $*/\omega$-operator found in ω-regular expressions. In contrast to LTL, RLTL allows to define arbitrary ω-regular properties, while keeping LTL's complexity of satisfiability (PSPACE). In contrast to νTL, RLTL refrains the user to deal with fix point formulas.

Technically, RLTL can be considered as a sublogic of linear fix point logic with chop (LFLC). As satisfiability for LFLC is undecidable, RLTL spots an interesting subset of LFLC. Moreover, practically, the techniques developed for LFLC [18, 15] should be usable for RLTL as well.

The careful reader has probably observed that complementation has not been included in RLTL, even though doing so does not immediately turn the decision problems non-elementary (regular-expressions would be built completely before complementation is applied). The reason is that complementation for ABA (using the translation from AcBA to WABA to obtain an ABA) involves a quadratic blow-up, and the resulting ABA for a given formula obtained at the end of the inductive construction will no-longer have a polynomial size in all cases.

A different accepting condition can be used, for example a parity condition, giving a linear size parity automaton at the end of the translation (using possibly a linear number of colors). One way to attack the emptiness problem of parity automata is then to translate the automaton into a *weak* alternating parity automaton, whose emptiness problem is well known to be in PSPACE. However, the best procedure known generates a weak automaton of size $O(n^k)$, for n states and k ranks). The use of weak parity automata directly in the construction is precluded by the dual power operator $z_x y$, since one seems to be forced to express that the initial state q_0 must be visited infinitely often.

Recent developments [23] seem to indicate that the emptiness problem for alternating parity automata is in PSPACE, by a direct algorithm. This result

would allow the introduction of negation in RLTL freely with no penalty in the complexity class of the algorithms.

Nevertheless, as the resulting alternating Büchi automaton for a given RLTL formula can be complemented with an at most quadratic blow-up, we can easily get an exponentially bigger nondeterministic Büchi automaton accepting the formula's refutations, so that automata-based model checking of RLTL specifications can be carried out as usual (for LTL).

Clearly, since ETL, νTL, DLTL, and RLTL are all expressively complete wrt. ω-regular languages, for every ground formula in one logic defining some ω-regular language, there is an equivalent ground formula in any of the other logics defining the same language. From that perspective, all logics are equally expressive. However, ETL, and νTL offer, for example, no translation of the sequencing operator ; respecting a given context and DLTL does not allow to formulate a corresponding ω-operator. Thus, RLTL's unique feature is that every LTL operator and the operators in regular expressions can be translated into an equivalent RLTL context (with the same number of *holes*). This allows a linear, *inductive* translation of LTL properties or regular expressions.

The closest approach to RLTL is DLTL, though it is motivated in the context of dynamic logic. Similarly as RLTL, DLTL implicitly follows similar concepts as *attempt*, *obligation*, and *delay* by enriching the until operator. However, the obligation must be met in every "intermediate" position between the current and the one where the attempt holds. In RLTL, however, a sequence of delays has to be considered and the obligation has to hold only when the delay begins. The choice taken in RLTL has a huge advantage: It is straightforward to extend RLTL with past operators, by changing the *direction* of delay expressions. Extending DLTL to handle past seems to be much more cumbersome. However, this addition is left for future work.

References

1. IEEE P1850 - Standard for PSL - Property Specification Language, Sep 2005.
2. Howard Barringer, Ruurd Kuiper, and Amir Pnueli. A really abstract concurrent model and its temporal logic. In *Procs. of the 13th Annual ACM Symp. on Principles of Programming Languages (POPL'86)*, pages 173–183, 1986.
3. Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic Sugar. In *Procs. of the 13th Int'l Conf. on Computer Aided Verification (CAV'01)*, pages 363–367. Springer, 2001.
4. Julius Richard Büchi. On a decision method in restricted second order arithmetic. In *Proc. of the Int'l Congress on Logic Methodology and Philosophy of Science*, pages 1–12. Stanford University Press, 1962.
5. E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. of the 7th Colloquium on Automata, Languages and Programming (ICALP'80)*, pages 169–181. Springer, 1980.
6. Dana Fisman, Cindy Eisner, and John Havlicek. *Formal syntax and Semantics of PSL: Appendix B of Accellera Property Language Reference Manual, Version 1.1*, March 2004.

7. David Harel and Doron Peleg. Process logic with regular formulas. *Theoretical Computer Science*, 38:307–322, 1985.
8. Jesper G. Henriksen and P. S. Thiagarajan. Dynamic linear time temporal logic. *Annals of Pure and Applied Logic*, 96(1–3):187–207, 1999.
9. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.
10. Juraj Hromkovic, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ϵ-free nondeterministic finite automata. In *Proc. of STACS'97*, volume 1200 of *LNCS*, pages 55–66. Springer, 1997.
11. Stephen C. Kleene. Representation of events in nerve nets and finite automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, volume 34, pages 3–41. Princeton University Press, Princeton, New Jersey, 1956.
12. Dexter Kozen. Results on the propositional μ-calculus. In *Proc. ICALP'82*, pages 348–359. Springer, 1982.
13. Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. In *Proc. of the Fifth Israel Symposium on Theory of Computing and Systems, ISTCS'97*, pages 147–158. IEEE Computer Society Press, 1997.
14. Martin Lange. Weak automata for the linear time μ-calculus. In *Procs. of the 6th Int'l Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 267–281. Springer, 2005.
15. Martin Lange and Colin Stirling. Model checking fixed point logic with chop. In *Procs. of the 5th Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *LNCS*. Springer, 2002.
16. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer, 1995.
17. David E. Muller and Paul E. Schupp. Altenating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.
18. Markus Müller-Olm. A modal fixpoint logic with chop. In *Proc. of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of *LNCS*, pages 510–520. Springer, 1999.
19. Amir Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–67, 1977.
20. Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems – a survey of current trends. In *Current Trends in Concurrency*, volume 224 of *LNCS*, pages 510–584. Springer, 1996.
21. A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear termporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.
22. Larry J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, Department of Electrical Engineering, MIT, Boston, Massachusetts, 1974.
23. Moshe Y. Vardi. Personal communication.
24. Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.
25. Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.
26. Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.
27. Hiroaki Yamamoto. On the power of input-synchronized alternating finite automata. In *Computing and Combinatorics: 6th Annual International Conference, COCOON 2000*, volume 1858 of *LNCS*, page 457. Springer, July 2000.