Regular Linear Temporal Logic with Past

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

[1] Madrid Institute for Advanced Studies (IMDEA Software), Spain
[2] Spanish Council for Scientific Research (CSIC), Spain
[3] Technische Universität München, Germany

Abstract. This paper upgrades Regular Linear Temporal Logic (RLTL) with past operators and complementation. RLTL is a temporal logic that extends the expressive power of linear temporal logic (LTL) to all ω-regular languages. The syntax of RLTL consists of an algebraic signature from which expressions are built. In particular, RLTL does not need or expose fix-point binders (like linear time μ-calculus), or automata to build and instantiate operators (like ETL$_*$).
Past operators are easily introduced in RLTL via a single previous-step operator for basic state formulas. The satisfiability and model checking problems for RLTL are PSPACE-complete, which is optimal for extensions of LTL. This result is shown using a novel linear size translation of RLTL expressions into 2-way alternating parity automata on words. Unlike previous automata-theoretic approaches to LTL, this construction is compositional (bottom-up). As alternating parity automata can easily be complemented, the treatment of negation is simple and does not require an upfront transformation of formulas into any normal form.

1 Introduction

In his seminal paper [23], Pnueli proposed Linear temporal logic (LTL) [20] as a specification language for reactive systems. LTL is a modal logic over a linear frame, whose formulas express properties of infinite traces using two *future* modalities: *nexttime* and *until*. Although extending LTL with past operators (e.g., [12]), does *not* increase its expressive power [8], it has been widely noticed that it caters for specifications that are shorter, easier and more intuitive [19]. For example,[17] shows that there is a family of LTL formulas with past operators whose equivalent future only formulas are exponentially larger. Likewise, recalling the classical example from [27], the specification that *Every alarm is due to a fault* can easily be expressed by $\Box(\mathit{alarm} \rightarrow \diamondminus \mathit{fault})$, where $\Box$ means *globally/always* and $\diamondminus$ means *once in the past*. An equivalent formulation using only future operators is $\neg(\neg \mathit{fault}\, \mathcal{U}\, (\mathit{alarm} \wedge \neg \mathit{fault}))$, which is, however, less intuitive. The problems of satisfiability and model checking are PSPACE-complete [17] for LTL with and without past operators, so the past does not seem to harm in terms of complexity.

With regards to expressivity, Wolper [32] showed that LTL cannot express all ω-regular properties. In particular, it cannot express the property "p holds only at even moments". In spite of being a useful specification language, this lack of expressivity seems to surface in practice [25]. To alleviate the expressivity

problem, Wolper suggested *extended temporal logic* (ETL) in which new operators are defined using automata, and instantiated using language composition. ETL was later extended [31, 14] to different kinds of automata. The main drawback of these logics is that, in order to obtain the full expressivity, an infinite number of operators is needed. Among other consequences for its practical usage, this implies that ETL is not algebraic. An alternative approach consists of adapting the modal μ-calculus [6, 13] to the linear setting (νTL) [1]. Here, the full expressivity is obtained by the use of fix point operators. In νTL one needs to specify recursive equations to describe temporal properties, since the only modality is *nexttime*, which tends to make typical specifications cumbersome.

At the same time, some studies [3] 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. Even though every ground regular expression can be translated into a νTL expression [15], the concatenation operator cannot be directly represented in νTL. No context of νTL can capture concatenation. Extending νTL with concatenation leads to *fix point logic with chop* (FLC) [22] that allows expressing non-regular languages, but at the price of undecidable satisfiability and equivalence problems.

Some dynamic logics also try to merge regular expressions (for the program part) with LTL (for the action part), for example, Regular Process Logic [9]. However, the satisfiability problem is non-elementary because one can combine arbitrarily negations and regular operators. Dynamic linear-temporal logic DLTL [10] (see also [16]) keeps the satisfiability problem in PSPACE, but restricts the use of regular expressions only as a generalization of the until operator. The until operator $p\mathcal{U}^{\alpha}\, q$ in DLTL is equipped with a regular expression (α) and establishes that the until part (q) must be fulfilled at some position in which α matches, while the first argument p must hold at all positions in between. It is unclear then how to extend DLTL with past operators. The approach of defining past operators using past regular expressions, presented in Section 2 for RLTL cannot be used for DLTL since the notion of "in-between" is not clear anymore. Another extension of LTL to regular expressions is the logic RELTL from [4]. However, this logic does not include past operators or negation. Moreover, it requires a translation into positive normal form for the LTL part that makes this translation not compositional. Also, the interaction of regular expressions and linear temporal logic in RELTL is restricted to prefixes, while in RLTL we consider more sophisticated combinations.

The popularity of regular expressions led also to their inclusion in the industry standard specification language PSL [7]. While decision procedures and their complexities for full PSL are still an area of active research, [16] shows that the fragment of PSL that contains LTL and semi-extended regular expressions, even though it allows more succinct specifications, leads to EXPSPACE-complete satisfiability and model checking problems, which may limit its practical applicability.

In this paper, we upgrade *Regular Linear Temporal Logic* (RLTL) [18] with past operators. RLTL is a temporal logic that extends the expressive power of LTL to all ω-regular languages. It has an algebraic signature and fuses LTL and regular expressions. To enrich RLTL by past operators, it suffices, as we show here,

to simply add *basic past expressions*, which allow the formulation of past regular expressions. Intuitively, regular expressions with past expressions can define finite segments of infinite strings in an arbitrary forward and backward manner. The main contribution of RLTL comes perhaps from the simplicity of the novel power operators, which allow the definition of most other temporal operators and, as we show here, the treatment of past and negation while avoiding non-algebraic constructs like fix-points bindings or automata instantiations. The power operators are the key to obtain compositionality without requiring an upfront translation to positive normal forms.

To address satisfiability and model checking for RLTL, we follow the automata theoretic approach, but need a more sophisticated translation than in [18] to cope with the new operators. This novel linear size translation uses 2-way alternating parity automata on words. Besides being useful for RLTL, this translation is also interesting for plain LTL, as it is compositional (bottom-up) unlike previous automata-theoretic approaches to LTL. As alternating parity automata can easily be complemented, the treatment of negation is simple and does not require an upfront transformation of formulas into positive or other normal form. A notable exception is [26], which presents another compositional translation from LTL, but this translation generates testers instead of automata.

Building on recent automata results [5], we show here that the satisfiability and model checking problems for RLTL (with past) are PSPACE-complete, which is optimal for extensions of LTL.

This paper is structured as follows. Section 2 introduces RLTL. Section 3 recalls the basic definitions of LTL with past, and presents the translation into RLTL. Section 4 describes the translation from RLTL into automata. Finally, Section 5 contains the conclusions. Due to space limitations some proofs are missing, but they can be easily reconstructed.

2 Regular Linear Temporal Logic

We define *regular linear temporal logic* (RLTL) in two stages, similarly to PSL or ForSpec. First, we present a variation of regular expressions enriched with a simple past operator. Then we use these regular expressions to define regular linear temporal logic as a language that describes sets of 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.

2.1 Regular Expressions with Past

We first introduce a variation of regular expressions with a past operator to describe finite segments of infinite words. The basic elements are *basic expressions*, which are Boolean combinations of a finite set of elementary propositions, interpreted in a single state (or in a single action between two states). Each set of propositions (or equivalently, each basic expression) can also be interpreted as a symbol from a discrete alphabet Σ that includes **true** (for all propositions) and **false** for the empty set or propositions.

Syntax The language of regular expressions for finite words is given by the following grammar:

$$\alpha ::= \alpha + \alpha \quad | \quad \alpha \,;\, \alpha \quad | \quad \alpha * \alpha \quad | \quad p \quad | \quad {}^{-}p$$

where p ranges over basic expressions. The intended interpretation of the operators $+$, ; and $*$ are the standard union, concatenation and binary Kleene-star. There is one expression of the form ${}^{-}p$ for each basic expression p. Informally, p indicates that the next "action", or input symbol, satisfies the basic expression p; similarly, ${}^{-}p$ establishes that the previous action or symbol satisfies p. Expressions of the form ${}^{-}p$ are called *basic past expressions*. Regular expressions are defined using an algebraic signature (symbols like p and ${}^{-}p$ are constants, and $+$, ; and $*$ are binary symbols).

Semantics Our version of regular expressions describe *segments* of infinite words. An infinite word w is a map from ω into Σ (i.e., an element of Σ^{ω}). A *position* is a natural number. We use $w[i]$ for the symbol at position i in word w. If $w[i]$ satisfies the basic expression p, we write $w[i] \vDash p$, which is defined in the standard manner. Given an infinite word w and two positions i and j, the tuple (w, i, j) is called the *segment* of the word w between positions i and j. It is not necessarily the case that $i < j$ or even that $i \leq j$. Note that a segment consists of the whole word w with two tags, not just the sequence of symbols that occur between two positions. A *pointed word* is a pair (w, i) formed by a word w and a position i. The semantics of regular expressions is formally defined as a binary relation $\vDash_{\text{RE}}$ between segments and regular expressions. This semantics is defined inductively as follows. Given a basic expression p, regular expressions x, y and z, and a word w:

- $(w, i, j) \vDash_{\text{RE}} p$ 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, $(w, i, k) \vDash_{\text{RE}} x$ 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 $(i_0 = i, i_1, \ldots i_m)$ and all $k \in \{0, .., m-1\}$ $(w, i_k, i_{k+1}) \vDash_{\text{RE}} x$ and $(w, i_m, j) \vDash_{\text{RE}} y$.
- $(w, i, j) \vDash_{\text{RE}} {}^{-}p$ whenever $w[j]$ satisfies p and $j = i - 1$.

One interesting expression using past is:

$$\mathit{notfirst} \stackrel{\text{def}}{=} {}^{-}\mathbf{true} \,;\, \mathbf{true}$$

which matches all segments of the form (w, i, i) that are not initial prefixes (i.e., $i \neq 0$). The semantics style used here is more conventional in logic than in automata theory, where regular expressions define sets of finite words. If one omits the basic past expressions, then 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 $w \in \Sigma^{\omega}$, $(vw, 0, |v|) \vDash_{\text{RE}} x$. Following this alternative interpretation, our operators correspond to the classical ones and regular expressions define precisely regular sets of non-empty words.

The following theorem shows that only a finite bounded amount of information is needed to determine whether a segment satisfies a regular expression. All

modified words that preserve all symbols within these bounds will contain a corresponding matching segment.

Theorem 1 (Relevant segment). *Let x be a regular expression and (w, i, j) a segment of an infinite word for which $(w, i, j) \vDash_{\text{RE}} x$. There exists bounds $A \leq i, j \leq B$ such that for every word prefix $v \in \Sigma^*$ and suffix $u \in \Sigma^\omega$, the infinite word $w' = vw[A, B]u$ satisfies:*

$$(w', |v| + (i - A), |v| + (j - A)) \vDash_{\text{RE}} x$$

Here, $w[A, B]$ is the finite word $w[A]w[A + 1] \cdots w[B]$.

Expressions that do not include basic past expressions ^-p are called future-only regular expressions and satisfy strict bounds: $A = i \leq j = B$.

Past Expressions In order to justify that basic past expressions allow to express conditions on the input symbols previously seen we introduce a new operator for regular expressions, by lifting basic past expressions into a past operator $(\cdot)^{-1}$:

$$\begin{aligned} (p)^{-1} &\stackrel{\text{def}}{=} {}^-p & (x + y)^{-1} &\stackrel{\text{def}}{=} x^{-1} + y^{-1} \\ ({}^-p)^{-1} &\stackrel{\text{def}}{=} p & (x\,;\,y)^{-1} &\stackrel{\text{def}}{=} y^{-1}\,;\,x^{-1} \\ & & (x * y)^{-1} &\stackrel{\text{def}}{=} y^{-1} + y^{-1}\,;\,(x^{-1} * x^{-1}) \end{aligned}$$

This definition is inductive, so every past expression can be transformed into an equivalent expression without $(\cdot)^{-1}$ (but perhaps with one or more ^-p).

We now study some properties of past expressions, justifying that $(\cdot)^{-1}$ is a good definition for a past construct. First, $(\cdot)^{-1}$ is its own self-inverse:

Lemma 1. *Every regular expression x is semantically equivalent to $(x^{-1})^{-1}$.*

Semantic equivalence means that both expressions define precisely the same set of segments. Intuitively, matching an expression x with a sequence of events should correspond to matching the past expression x^{-1} with the *reversed* sequence of events. Since input words are infinite only on one end, this intuition is not justified simply by reversing the linear order of symbols in an infinite word. The following theorem formalizes this intuition of reverse by providing an evidence of a finite portion of input that can be chopped and reversed to match the inverse expression.

Theorem 2 (Inverse and reverse). *Let x be a regular expression and (w, i, j) a segment of an infinite word for which $(w, i, j) \vDash_{\text{RE}} x$. There exists bounds $A \leq i, j \leq B$ for which for all prefix $v \in \Sigma^*$ and suffix $u \in \Sigma^\omega$, the infinite word $w' = vw[A, B]^{rev}u$ satisfies:*

$$(w', |v| + (B - j), |v| + (B - i)) \vDash_{\text{RE}} x^{-1}$$

Here, $w[A, B]^{rev}$ is the finite word $w[B]w[B - 1] \cdots w[A]$, the reverse of $w[A, B]$.

Finally, the following theorem justifies that if an expression x matches some input, then the concatenation of x with its inverse x^{-1} must match the segment that goes back to the initial position.

Theorem 3 (Inverse and sequential). *Let x be a regular expression and (w, i, j) a segment for which $(w, i, j) \vDash_{\text{RE}} x$. Then $(w, i, i) \vDash_{\text{RE}} x\,;\,x^{-1}$*

2.2 Regular Linear Temporal Logic over Infinite Words

Regular Linear Temporal Logic expressions denote languages over infinite words. The key elements of RLTL are the two *power* operators that generalize many constructs from different linear-time logics and calculi.

Syntax The syntax of RLTL expressions is defined by the following grammar:

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

where α ranges over regular expressions. Informally, $\vee$ stands for union of languages (disjunction in a logical interpretation), and $\neg$ represents language complement (or negation in a logical framework). The symbol ; stands for the conventional concatenation of an expression over finite words followed by an expression over infinite words. The operator $\varnothing$ represents the empty language (or *false* in a logical interpretation).

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 is captured by $p\,|\mathbf{true}\,;\mathbf{true}\rangle\mathbf{false}$.

Note that the signature of RLTL is, like that of RE, 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 RE and 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 $\models$ between expressions and pointed words, defined inductively. Given two RLTL expressions x and y, a regular expression z, and a word w:

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

The semantics of $x|z\rangle\!\rangle y$ 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 z—after which y holds, and x holds after each individual delay. The semantics of $x\,|z\rangle y$ also allow the case where y never holds, but x always holds after any number of evaluations of z. 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. *For every RLTL expressions x and y and* RE *expression z:*
– $x|z\rangle\!\rangle y$ *is semantically equivalent to* $y \vee (x \wedge z\,;x|z\rangle\!\rangle y)$.
– $x\,|z\rangle y$ *is semantically equivalent to* $y \vee (x \wedge z\,;x\,|z\rangle y)$.

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 $x|z\rangle\!\rangle y$ is then characterized to a least fix point, while $x\,|z\rangle y$ is characterized by a greatest fix-point.

Remark 1. It should be noted that although RLTL includes complementation it does not allow the use of complementation within regular expressions. It is well-known [29] 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 [16].

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. Pumping was a primitive operator in [18], for a simpler logic without negation. To ease the translation from LTL into RLTL presented in the next section we introduce some RLTL syntactic sugar:

$$\top \stackrel{\text{def}}{=} \neg\varnothing \qquad \mathit{first} \stackrel{\text{def}}{=} \neg(\mathit{notfirst}\,;\top)$$

3 LTL with Past

In this section we show how to translate LTL (past and future) into RLTL. Unlike in [18], the translation presented here does not require a previous transformation of LTL expressions into their negation normal form. The translation is purely linear: every LTL operator corresponds to an RLTL context with the same number of "holes".

We consider the following minimal definition of LTL, with an interpretation of atomic propositions as actions. Given a finite set of propositions *Prop* (with p a representative) called basic action expressions, the language of LTL expressions given by the following grammar:

$$\psi ::= p \;\mid\; \psi \vee \psi \;\mid\; \neg\psi \;\mid\; \bigcirc\psi \;\mid\; \psi\,\mathcal{U}\,\psi \;\mid\; \ominus\psi \;\mid\; \psi\,\mathcal{B}\,\psi$$

Here, $\neg$ and $\vee$ are the conventional Boolean expressions. The operators $\bigcirc$, and $\mathcal{U}$ are the future operators. Finally, $\ominus$ and $\mathcal{B}$ are called past operators.

Informal semantics LTL expressions define sets of pointed words. A pointed word (w, i) satisfies a basic action expression p if action p is taken from $w[i]$. Boolean operators are interpreted in the conventional way. An expression $\bigcirc x$ (read *next* x) indicates that in order for a pointed word (w, i) to satisfy $\bigcirc x$ its sub-expression x must hold when interpreted at the next position: $(w, i+1)$. Similarly, $\ominus x$ (read *previous* x) holds at (w, i) if x holds at $(w, i-1)$ or i is the initial position $(w, 0)$. The operator $x\mathcal{U}\,y$ (read x *until* y) holds at (w, i) whenever y holds at some future position and x holds in all positions in between. Similarly, $x\,\mathcal{B}\,y$ (read x *back-to* y) states that x holds in all previous positions (including the present) starting at the last position y held (or from the initial position 0 if y does not hold in any past position).

Semantics The semantics of LTL expressions is defined inductively. Let p be a basic expression, and x and y be arbitrary LTL expressions.

- $(w, i) \vDash_{\mathrm{LTL}} p$ whenever $w[i]$ satisfies p.
- $(w, i) \vDash_{\mathrm{LTL}} x \vee y$ whenever $(w, i) \vDash_{\mathrm{LTL}} x$ or $(w, i) \vDash_{\mathrm{LTL}} y$.
- $(w, i) \vDash_{\mathrm{LTL}} \bigcirc x$ whenever $(w, i+1) \vDash_{\mathrm{LTL}} x$.
- $(w, i) \vDash_{\mathrm{LTL}} x\,\mathcal{U}\,y$ whenever for some $j \geq i$, $(w, j) \vDash_{\mathrm{LTL}} y$, and $(w, k) \vDash_{\mathrm{LTL}} x$ for all $i \leq k < j$.
- $(w, i) \vDash_{\mathrm{LTL}} \ominus x$ whenever either $i = 0$ or $(w, i-1) \vDash_{\mathrm{LTL}} x$.
- $(w, i) \vDash_{\mathrm{LTL}} x\,\mathcal{B}\,y$ whenever $(w, j) \vDash_{\mathrm{LTL}} x$ for all $j \leq i$, or for some $k \leq i$, $(w, k) \vDash_{\mathrm{LTL}} y$ and for all l within $k < l \leq i$, $(w, l) \vDash_{\mathrm{LTL}} x$.

We now show how to translate LTL expressions into RLTL. First, we define recursively a map between LTL expressions and RLTL expressions and then prove that each LTL expression is equivalent to its image.

$$\begin{array}{ll} f(p) = p & f(\bigcirc x) = \mathbf{true}\,;\, f(x) \\ f(x \vee y) = f(x) \vee f(y) & f(x\,\mathcal{U}\,y) = f(x)\,|\mathbf{true}\rangle\!\rangle f(y) \\ f(\neg x) = \neg f(x) & f(\ominus x) = \mathit{first} \vee {}^{-}\mathbf{true}\,;\, f(x) \\ & f(x\,\mathcal{B}\,y) = f(x)\,|{}^{-}\mathbf{true}\rangle f(y) \end{array}$$

The function $f(\cdot)$ is well-defined by construction. Since both LTL and RLTL expressions define sets of pointed words equivalence $\equiv$ is simply equality between two sets of pointed words.

Theorem 4. *Every LTL expression is equivalent to its RLTL translation.*

A practical specification language based on LTL offers more operators than the minimal set presented above, including other Boolean connectives and additional future operators like $\Box x$ (*always* x or *henceforth* x), $\Diamond x$ (read *eventually* x), $y \mathcal{R} x$ (y *release* x), etc. Additional past operators include $\ominus x$ (a strong version of $\circleddash x$), $\boxminus x$ (*has always been* x), $\diamondminus x$ (*once* x), $x \mathcal{S} y$ (read x *since* y), etc. All these can be defined in terms of the minimal set using the following LTL equivalences [20]:

$$\begin{array}{ll} \Diamond x \equiv \mathbf{true}\, \mathcal{U}\, x & x \,\mathcal{R}\, y \equiv \neg(\neg y \mathcal{U} \neg x) \\ \Box x \equiv \neg\Diamond\neg x & x \,\mathcal{W}\, y \equiv (x \,\mathcal{U}\, y) \vee \Box x \\ \boxminus x \equiv x \,\mathcal{B}\, \mathbf{false} & \ominus x \equiv \neg\circleddash\neg x \\ \diamondminus x \equiv \neg\boxminus\neg x & x \,\mathcal{S}\, y \equiv (x \,\mathcal{B}\, y) \wedge \diamondminus y \end{array}$$

Proceeding with these equivalences, however, does not generate an LTL expression (and consequently a RLTL expression) of linear size. In particular $\mathcal{W}$ and $\mathcal{S}$ duplicate one of their parameters. A formula with a stack of nested $\mathcal{W}$ or $\mathcal{S}$ symbols will generate an exponentially larger formula. On the contrary, the following direct translations into RLTL are linear:

$$f(x \,\mathcal{W}\, y) = f(x)\, |\mathbf{true}\rangle f(y) \qquad f(x \,\mathcal{S}\, y) = f(x) |^{-}\mathbf{true}\rangle\!\rangle f(y)$$

The translation function f only involves a linear expansion in the size of the original formula. Since checking satisfiability of linear temporal logic is PSPACE-hard [28] this translation implies 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 From RLTL to Automata

We now show how to translate an RLTL expression into a 2-way Alternating Parity Automaton of linear size that accepts precisely the same set of pointed words. As we justify below this implies that the problems of emptiness and model checking for RLTL are in PSPACE.

Preliminaries Let us first present the necessary definitions of non-deterministic automata on finite words and alternating automata on infinite words.

A 2-*way nondeterministic finite automaton* (2NFA) is a tuple $\mathcal{A} : \langle \Sigma, Q, q_0, \delta, F \rangle$ where Σ is the alphabet, Q a finite set of *states*, $q_0 \in Q$ the *initial state*, $\delta : Q \times \Sigma \to 2^{Q \times \{-1,0,1\}}$ the *transition function*, and $F \subseteq Q$ is the set of *final states*. Intuitively, the automaton works by reading an input tape. The transition function indicates the legal moves from a given state and character in the tape. A transition is a successor state and the direction of the head of the tape. Our

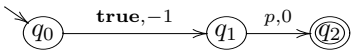
version of 2NFA operates on segments of infinite words. A *run* of $\mathcal{A}$ on a word $w \in \Sigma^\omega$, starting at position i_0 and finishing at position i_n is a sequence of states and positions $i_0 q_0 i_1 q_1 i_1 \dots i_n q_n$, where q_0 is the initial state of $\mathcal{A}$, and for all $k \in \{1, \dots n\}$ we have that $(q_k, i_k - i_{k-1}) \in \delta(q_{k-1}, w[i_{k-1}])$. The run is called *accepting* if $q_n \in F$. A 2NFA accepts a segment (w, i, j) whenever there is an accepting run starting at i and finishing at j. There is an immediate correspondence to regular expressions:

Lemma 3. *Each regular expression can be translated into an equivalent* 2NFA.

In the proof of Lemma 3 the translation from regular expressions into 2NFA follows the standard bottom-up construction used for conventional regular expressions into NFA [11] for the operators ;, $*$ and $+$, and the basic expressions p. The translation of basic past expression $^- p$ is the automaton: $\langle \Sigma, \{q_0, q_1, q_2\}, q_0, \delta, \{q_2\}\rangle$ with

$$\delta(q_0, \mathbf{true}) = \{(q_1, -1)\}, \quad \delta(q_1, p) = \{(q_2, 0)\}, \quad \delta(q_2, true) = \{\},$$

depicted graphically:



This translation clearly coincides with the semantics of $^- p$. The number of states of the 2NFA obtained is linear in the size of the regular expression.

We define now alternating automata on infinite-words. 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 $\varphi, \psi \in \mathcal{B}^+(\mathcal{X})$ implies $\varphi \wedge \psi \in \mathcal{B}^+(\mathcal{X})$ and $\varphi \vee \psi \in \mathcal{B}^+(\mathcal{X})$. We say that a set $Y \subseteq \mathcal{X}$ *satisfies* (or is a *model* of) a formula $\varphi \in \mathcal{B}^+(\mathcal{X})$ 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 $(q_1 \vee q_2) \wedge q_3$. The dual of a formula $\theta \in \mathcal{B}^+(\mathcal{X})$ is the formula $\overline{\theta} \in \mathcal{B}^+(\mathcal{X})$ obtained by exchanging **true** and **false**, and $\wedge$ and $\vee$.

A 2-*way Alternating Parity Automaton on Words* (2APW) is a tuple $\mathcal{A} : \langle \Sigma, Q, q_0, \delta, F\rangle$ where Σ, Q are as for 2NFA. The transition function δ yields a positive Boolean combination of successor states, together with a direction: $\delta : Q \times \Sigma \to \mathcal{B}^+(Q \times \{-1, 0, 1\})$. The acceptance condition F that we use here is the *parity* acceptance condition:

$$F : Q \to \{0 \dots k\}.$$

The set $\{0 \dots k\}$ is called the set of colors. A 2APW operates on infinite words: a *run* over an infinite word $w \in \Sigma^\omega$ is a directed graph (V, E) such that $V \subseteq Q \times \mathbb{N}$ satisfying the following properties:

1. $(q_0, 0)$ is in V, and it is called the initial vertex. It may have no predecessor.
2. every non-initial vertex has a predecessor. For every (q, l) distinct from $(q_0, 0)$

$$\{(q', l') \in V \mid (q', l') \to_E (q, l)\} \neq \emptyset$$

3. the successors of every node form a minimal model for δ, i.e., for every vertex (q, l), the set $\{(q', l' - l) \mid (q, l) \rightarrow_E (q', l')\}$ is a minimal model of $\delta(q, w[l])$.

The set of vertices that occurs infinitely often in an infinite path π is denoted $inf(\pi)$. A run (V, E) is *accepting* according to F if every maximal finite path ends in a vertex (q, l) with $\delta(q, w[l]) = \mathbf{true}$ and every infinite path π accepts the parity condition:

$$max\{i \mid i = F(q) \text{ for some } q \text{ in } inf(\pi)\} \text{ is even.}$$

The language $\mathcal{L}(\mathcal{A})$ of a 2APW $\mathcal{A}$ is determined by all strings for which an accepting run of $\mathcal{A}$ exists. We measure the *size* of a 2APW in terms of its number of states and its number of colors.

4.1 Complementing 2APW

Every 2APW $\mathcal{A}$ can be easily complemented into another 2APW $\overline{\mathcal{A}}$ of the same size. Let n be the number of states of $\mathcal{A}$. The key observation is that $\mathcal{A}$ can be transformed into an equivalent automaton with a color set $\{0 \dots k\}$ satisfying $k \leq n + 1$, by only changing the acceptance condition.

Let F be the acceptance condition for $\mathcal{A}$, and let F' be another acceptance condition such that,

Acc1. for every two nodes p and q, if $F(p) \leq F(q)$ then $F'(p) \leq F'(q)$.
Acc2. for every node p, $F(p)$ is even if and only if $F'(p)$ is even.

Then, given a path π of a run of $\mathcal{A}$, if q is a node occurring infinitely often with maximum color according to F, then q is also maximum according to F'. Moreover, $F(q)$ is even if and only if $F'(q)$ is even. Therefore, every run of $\mathcal{A}$ is accepting according to F if and only if it is also accepting according to F'.

Consequently, the following *gap reduction* procedure can be applied. Assume for some color i there is no node q with $F(q) = i$, but for some $j < i$ and for some $k > i$, there are such nodes $F(q_j) = j$ and $F(q_k) = k$. Color i is called a gap in F. The following F' is equivalent to F according to the conditions (Acc1) and (Acc2) described above:

$$F'(q) = \begin{cases} F(q) & \text{if } F(q) < i \\ F(q) - 2 & \text{if } F(q) > i \end{cases}$$

Similarly, if for no node q, $F(q) = 0$ or $F(q) = 1$, then an equivalent F' can be defined as $F'(q) = F(q) - 2$ for all q. By applying these transformations until no gap exists we ensure that all assigned colors are consecutive, and starting either at 0 or at 1. We use F^* to denote the accepting condition obtained after repeatedly applying the gap reduction procedure. It follows that the maximum color assigned by F^* can be at most $n + 1$. This property ensures the following lemma.

Lemma 4. *Every* 2APW *can be complemented into another* 2APW *of the same number of states and with highest color at most* $n + 1$.

Proof (Sketch). Let $\mathcal{A}$ be a 2APW. The following 2APW accepts the complement language:

$$\overline{\mathcal{A}} : \langle \Sigma, Q, q_0, \overline{\delta}, \overline{F}^* \rangle$$

where $\overline{\delta}(q, a)$ is the dual of the transition $\delta(q, a)$ and $\overline{F}(q) = F(q) + 1$, with $\overline{F}^*$ be the gap reduced version of $\overline{F}$. The maximum color in $\overline{F}^*$ is guaranteed to be at most $n+1$ (also at most 1 plus the number of colors in F^*). It is well-known [21] that the dualization of the transition function and acceptance condition satisfies that $\mathcal{L}(\mathcal{A}) = \Sigma^\omega \setminus \mathcal{L}(\overline{\mathcal{A}})$. □

4.2 Translating from RLTL to 2APW

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

Theorem 5. *For every RLTL formula φ, there is a* 2APW *with size linear in the size of φ that accepts precisely the same set of ω-words.*

The proof proceeds according to the following translation from RLTL into 2APW. The procedure works bottom-up the parse tree of the RLTL expression φ, building the resulting automaton using the subexpressions' automata as components. Our translation does not require an upfront transformation into negation normal form. On the contrary, it is truly compositional in a bottom-up fashion. The automaton for an expression is built from the automata of its subexpressions with all the structure preserved.

For RLTL expressions x, y and a regular expression z let $\mathcal{A}_x : \langle \Sigma, Q^x, q_o^x, \delta^x, F^x \rangle$ and $\mathcal{A}_y : \langle \Sigma, Q^y, q_o^y, \delta^y, F^y \rangle$ be two 2APW automata equivalent to x and y, and let $\mathcal{A}_z : \langle \Sigma, Q^z, q_o^z, \delta^z, F^z \rangle$ be a 2NFA for z. Without loss of generality, we assume that their state spaces are disjoint, and that the coloring is minimal ($F^x = (F^x)^*$ and $F^y = (F^y)^*$) . We consider the different operators of RLTL:

- **Empty**: The automaton for $\varnothing$ is $\mathcal{A}_\varnothing : \langle \Sigma, \{q_0\}, q_0, \delta, F \rangle$ with $\delta(q_0, a) = \mathbf{false}$ for every a, and $F(q_0) = 0$ (any number works here). Clearly, the language of $\mathcal{A}_\varnothing$ is empty.
- **Disjunction**: The automaton for $x \vee y$ is:

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

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

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

  For F, we consider the union of the characteristic graph of the function:

  $$F(q) = \begin{cases} F^x(q) & \text{if } q \text{ is in } Q^x \\ F^y(q) & \text{if } q \text{ is in } Q^y \\ min\{F^x(\cdot), F^y(\cdot)\} & \text{if } q = q_0 \end{cases}$$

  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 of the languages of x and y.

– **Complementation**: The automaton for $\neg x$ is:

$$\mathcal{A}_{\neg x} \ : \ \langle \Sigma, Q^x, q_0^x, \overline{\delta}, \overline{F^x}^* \rangle$$

where $\overline{\delta}$ and $\overline{F^x}^*$ is as defined in Lemma 4, which guarantees that the language for $\mathcal{A}_{\neg x}$ is the complement of that of $\mathcal{A}_x$.

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

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

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

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

and, for $q \in Q^x$ as $\delta(q,a) = \delta^x(q,a)$. Recall that $\mathcal{A}_z$ is a 2NFA automaton. The accepting condition is $F(q) = F^x(q)$ for q in Q^x, and $F(q) = 1$ for q in Q^z ensuring that looping forever in z is not a satisfying path. Whenever $\mathcal{A}_z$ can non-deterministically choose a successor that is a final state, it can also move to the initial state of $\mathcal{A}_x$. Thus, the accepted language is indeed the concatenation.

– **Power**: The automaton for $x|z\rangle\!\rangle y$ is:

$$\mathcal{A}_{x|z\rangle\!\rangle y} \ : \ \langle \Sigma, Q^z \cup Q^x \cup Q^y \cup \{q_0\}, q_0, \delta, F \rangle$$

where the initial state q_0 is a fresh state. The transition function δ is defined as follows. The a successor of q_0 is:

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

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

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

The construction follows precisely the equivalence $x|z\rangle\!\rangle y \equiv y \vee (x \wedge z; x|z\rangle\!\rangle y)$ established in Lemma 2 and the construction for disjunction, conjunction, and concatenation. Finally, the looping in z is prevented by assigning $F(q) = 1$ whenever q is in Q^z, and otherwise $F(q) = F^x(q)$ or $F(q) = F^y(q)$ whenever q is in Q^x (resp. Q^y). Finally, $F(q_0) = 1$ to ensure that an infinite path that traverses only states from Q^z and q_0 is not accepting.

– **Weak power**: The automaton for $x\,|z\rangle y$ is:

$$\mathcal{A}_{x|z\rangle y} \ : \ \langle \Sigma, Q^z \cup Q^x \cup Q^y \cup \{q_0\}, q_0, \delta, F \rangle$$

where q_0 and δ are like for Power. The states in Q^y and Q^x are mapped to the same colors, as before. Now, $F(q_0) = 2$, and $F(q) = 1$ for all q in Q^z. Then, a path that accepts z and visits q_0 infinitely often is accepting.

Complexity From Lemma 3 every regular expression can be translated into a 2NFA with only a linear blow-up in size. Each of the steps in the procedure for translating RLTL expressions into a 2APW add at most one extra state. Therefore, the number of states in the produced automaton is at most the number of symbols in the original expression. For the colors, the only construct that increases the number of colors is complementation. The rest of the constructs use constant colors (1 and 2), or the union of sets of colors. Therefore, the highest color in a generated automaton corresponds to the largest number of nested negations $\neg$ in the starting expression.

Second, the structure of the sub-automata is preserved in all stages. We do not use automata constructions like product or subset constructions; instead only new states and transitions are added. For the accepting condition, all operations preserve the accepting condition of the automata corresponding to the sub-expression, except for complementation. Observe also how the automaton for $\neg\neg x$ is exactly the same automaton as for x.

Given a 2APW with n states and k colors one can generate on-the-fly successor states and final states of an equivalent 1-way nondeterministic Büchi automaton on words (NBW) with $2^{O((nk)^2)}$ states [5]. Since emptiness of NBW can be checked in NLOGSPACE via reachability [30], it follows that emptiness of 2APW is in PSPACE. Hence, the satisfiability, equivalence and model checking problems for RLTL are in PSPACE. Together with Proposition 1:

Corollary 1. *Checking satisfiability of an RLTL formula is PSPACE-complete.*

Using clever manipulation of the automata during the bottom-up construction one can show that only 3 colors are needed, leading to a better translation into NBW than the one presented in this paper, using only $2^{O(n^2)}$ states. The detailed explanation of this advanced translation is out of the scope of this paper.

5 Conclusion and Future Work

Amir Pnueli postulated in [24]: *"In order to perform compositional specification and verification, it is convenient to use the past operators but necessary to have the full power of ETL"*. In this paper, we have introduced *regular linear temporal logic* (RLTL) with past operators that exactly fulfills Pnueli's requirements, while at the same time keeping satisfiability and model checking in the same complexity class as for LTL (PSPACE). RLTL (with past) has a finite set of temporal operators giving it a temporal logic flavor and allows the integration of regular expressions. Moreover, we have introduced a novel translation of RLTL formulas into corresponding automata, which may be of its own interest, as it is truly compositional (bottom-up).

It should be stressed that a practically relevant specification language needs a variety of different operators as well as macros to support engineers in the complex job of specifying requirements. In fact, together with industrial partners, the second author was involved in the development of the language SALT [2] which acts as a high-level specification language offering a variety of different constructs

while at the same time allowing a translation to LTL. However, the lack of regular expressions and past operators makes such a translation difficult, error prone, and leads to automata that do not reflect the structure of the original formula and might be larger than necessary. It is therefore essential to have a core logic that is expressive and allows a simple, verifiable translation to automata and allows a simple translation from high-level languages like SALT. We consider RLTL to exactly meet this goal. As future work, it remains to build corresponding satisfiability and model checking tools to push RLTL into industrial applications. Also, some of the operators in PSL can already be mapped into RLTL. For example, "*whenever* α *is matched* p *must be true*" can be expressed as $\neg(\alpha\ ;\neg p)$. The blow-up in complexity in PSL (EXPSPACE) with respect to RLTL (PSPACE) can then fully blamed to the availability of semi-extended regular expressions. Moreover, the sequential connective in PSL that connects a temporal operator with a regular expression requiring the overlap of the last symbol can be easily expressed in RLTL as $(z\ ;\ \mathbf{true}^{-1}\ ;\ x)$, which coincides with the PSL semantics, for future regular expressions. Future study include other PSL operators like bounded iteration and abort.

Another interesting line of future research is to study symbolic model-checking algorithms for RLTL.

Acknowledgements. We wish to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *POPL'86*, 1986.
2. A. Bauer, M. Leucker, and J. Streit. SALT—structured assertion language for temporal logic. In *ICFEM'06*, *LNCS 4260*, September 2006.
3. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *CAV'01*. Springer, 2001.
4. Doron Bustan, Alon Flaisher, Orna Grumberg, Orna Kupferman, and Moshe Y. Vardi. Regular vacuity. In *Proc. of the 13th IFIP WG 10.5 Advanced Research Working Conference (CHARME'05)*, *LNCS 3725*. Springer, 2005.
5. C. Dax and F. Klaedtke. Alternation elimination by complementation. In *LPAR'08*, *LNCS 5530*. Springer, 2008.
6. A. Emerson and E. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *ICALP'80*. Springer, 1980.
7. D. Fisman, C. Eisner, and J. Havlicek. *Formal syntax and Semantics of PSL: Appendix B of Accellera Property Language Reference Manual, Version 1.1*, 2004.
8. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal basis of fairness. In *POPL'80*, 1980.
9. D. Harel and D. Peleg. Process logic with regular formulas. *TCS*, 38:307–322, 1985.
10. J. Henriksen and P.S. Thiagarajan. Dynamic linear time temporal logic. *Annals of Pure and Applied Logic*, 96(1–3):187–207, 1999.
11. J. Hopcroft and J. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.
12. H. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, UCLA, 1968.

13. D. Kozen. Results on the propositional μ-calculus. In *ICALP'82*. Springer, 1982.
14. O. Kupferman, N. Piterman, and M. Vardi. Extended temporal logic revisited. In *CONCUR'01*, *LNCS 2154*. Springer, 2001.
15. M. Lange. Weak automata for the linear time μ-calculus. In *VMCAI'05*, *LNCS 3385*. Springer, 2005.
16. M. Lange. Linear time logics around PSL: Complexity, expressiveness, and a little bit of succinctness. In *CONCUR'07*, 2007.
17. F. Laroussinie, N. Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In *LICS'02*, 2002.
18. M. Leucker and C. Sánchez. Regular linear temporal logic. In *ICTAC'07*, *LNCS 4711*. Springer, 2007.
19. O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logic of Programs*, 1985.
20. Z. Manna and A. Pnueli. *Temporal Verif. of Reactive Systems*. Springer, 1995.
21. D. Muller and P. Schupp. Altenating automata on infinite trees. *TCS*, 54:267–276, 1987.
22. M. Müller-Olm. A modal fixpoint logic with chop. In *STACS'99*, *LNCS 1563*, 1999.
23. A. Pnueli. The temporal logic of programs. In *FOCS'77*, 1977.
24. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, *NATO ASI F-13*. Springer, 1985.
25. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems–a survey of current trends. In *Current Trends in Concurrency*, *LNCS 224*, 1996.
26. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *Proc. of 14th Int'l Symp. on Formal Methods (FM'2006)*, *LNCS 4085*. Springer, 2006.
27. Ph. Schnoebelen. The complexity of temporal logic model checking. In *AiML'02*, 2002.
28. A. P. Sistla and E. Clarke. The complexity of propositional linear termporal logics. *JACM*, 32(3):733–749, 1985.
29. L. Stockmeyer. *The Computational Complexity of Word Problems*. PhD thesis, MIT, 1974.
30. M. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, *LNCS 1043*. Springer, 1996.
31. M. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comp.*, 115:1–37, 1994.
32. P. Wolper. Temporal logic can be more expressive. *Info&Control*, 56:72–99, 1983.