Regular Linear-time Temporal Logic

Martin Leucker
Institut für Informatik
TU München
85748 Garching, Germany
Email: leucker@in.tum.de

César Sánchez
The IMDEA Software Institute, and
Spanish Council for Scientific Research (CSIC)
Madrid, Spain
Email: cesar.sanchez@imdea.org

***Abstract*—This extended abstract presents the gist of regular linear-time temporal logic (RLTL), a logic that generalizes linear-time temporal logic (LTL) with the ability to use regular expressions arbitrarily as sub-expressions. Unlike LTL, RLTL can define all ω-regular languages and unlike previous approaches, RLTL is defined with an algebraic signature, does not depend on fix-points in its syntax, and provides past operators 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.**

***Keywords*-temporal logic; LTL; regular expressions;**

I. Introduction

In his seminal paper [1], Pnueli proposed Linear temporal logic (LTL) [2] 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 modalities: *nexttime* and *until*. Although extending LTL with past operators (e.g., [3]), does *not* increase its expressive power [4], it has been widely noticed that it caters for specifications that are shorter, easier, and more intuitive [5]. LTL is a widely accepted formalism for the specification and verification of concurrent and reactive systems. The problems of satisfiability and model checking are PSPACE-complete [6] for LTL with and without past operators.

However, with regards to expressivity, Wolper [7] showed that LTL cannot express all ω-regular properties. 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 [8]. 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 [9], [10] 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 [11], [12] to the linear setting (νTL) [13]. 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 [14] 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) [16] 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 [17]. However, the satisfiability problem is non-elementary because one can combine arbitrarily negations and regular operators. Dynamic linear-temporal logic DLTL [18] keeps the satisfiability problem in PSPACE, but restricts the use of regular expressions only as a generalization of the until operator. It is unclear how to extend DLTL by corresponding past operators—like temporal past operators and past regular expressions.

The popularity of regular expressions led also to their inclusion in the industry standard specification language PSL [19]. While decision procedures and their complexities for full PSL are still an area of active research, [20] shows that the fragment of PSL that contains LTL and semi-extended regular expressions leads to EXPSPACE-complete satisfiability and model checking problems, which may limit its practical applicability.

In the remainder of this extended abstract, we try to give a flavor of RLTL. It is deliberately informal, as more and precise information on RLTL is published in [21], [22].

II. RLTL

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

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

where $x \mid y$ denotes the intersection operator present in (semi-)extended regular expressions [23]. 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,

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

The key elements of RLTL are (two) ternary *power* operators that incorporate the three elements *attempt*, *obligation*, and *delay*. The delay parameter of a power operator is expressed using a regular expression, while the attempt and the obligation are defined with arbitrary RLTL expressions.

Consequently, RLTL is defined in two stages: First, regular expressions enriched with a simple past operator are introduced. Then, based on these expressions, RLTL is defined as a language that describes sets of infinite words.

1) Regular Expressions with Past: Regular expressions are built on *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) and form the underlying alphabet Σ that includes $\mathbf{true}$ (for all propositions) and $\mathbf{false}$ for the empty set or propositions.

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

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

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*.

One interesting expression using past is:

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

which matches all segments of a word that are not initial prefixes.

2) Regular Linear Temporal Logic over Infinite Words: The syntax of RLTL expressions is then 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. In other words, for every RLTL expressions x and y and regular 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 $y \vee (x \wedge z\,;x\,|z\rangle y)$.

which captures both the patterns of the binary Kleene-star and of the until operator.

Past expressions are not within RLTL's minimal set of operators but are easily defined using past regular expressions. For example, weak-previously x, denoted $\ominus x$, is given by $\mathit{first} \vee {}^{-}\mathbf{true}$; x and x back to y ($x\,\mathcal{B}\,y$) is defined as $x\,|{}^{-}\mathbf{true}\rangle y$, where $\mathit{first} = \neg\mathit{notfirst}$.

III. From RLTL to Automata

The problems of emptiness and model checking for RLTL are PSPACE complete. This is shown by a translation an RLTL expression into a 2-way Alternating Parity Automaton of linear size that accepts precisely the same set of words. Since checking emptiness of 2APW is PSPACE complete [24], it follows that satisfiability of RLTL is PSPACE complete as well. We briefly sketch that translation.

Like the definition of RLTL, the translation is done in two stages. First, we state that each regular expression can be translated into an equivalent 2-*way nondeterministic finite automaton* 2NFA. Intuitively, a 2NFA 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. A particularity in the context of RLTL is that our version of 2NFA operates on finite segments of infinite words.

In the second stage, an RLTL formula is translated into a corresponding 2-*way Alternating Parity Automaton on Words* (2APW). The transition function of a 2APW yields a positive Boolean combination of successor states (rather than a set of possible successor states in a non-deterministic automaton), together with a direction, like in 2NFA. The procedure translating an RLTL formula works bottom-up the parse tree of the RLTL expression, building the resulting automaton using the subexpressions' automata as components.

REFERENCES

[1] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, 1977, pp. 46–67.

[2] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems*. Springer-Verlag, 1995.

[3] H. W. Kamp, "Tense logic and the theory of linear order," Ph.D. dissertation, University of California, Los Angeles, 1968.

[4] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, "On the temporal basis of fairness," in *POPL*, 1980, pp. 163–173.

[5] O. Lichtenstein, A. Pnueli, and L. D. Zuck, "The glory of the past," in *Logic of Programs*, ser. Lecture Notes in Computer Science, R. Parikh, Ed., vol. 193. Springer, 1985, pp. 196–218.

[6] F. Laroussinie, N. Markey, and P. Schnoebelen, "Temporal logic with forgettable past," in *Proc. IEEE Symp. Logic in Computer Science (LICS'2002)*, 2002, pp. 383–392. [Online]. Available: citeseer.nj.nec.com/laroussinie02temporal.html

[7] P. Wolper, "Temporal logic can be more expressive," *Information and Control*, vol. 56, pp. 72–99, 1983.

[8] A. Pnueli, "Applications of temporal logic to the specification and verification of reactive systems – a survey of current trends," in *Current Trends in Concurrency*, ser. Lecture Notes in Computer Science, vol. 224. Springer-Verlag, 1996, pp. 510–584.

[9] M. Y. Vardi and P. Wolper, "Reasoning about infinite computations," *Information and Computation*, vol. 115, pp. 1–37, 1994.

[10] O. Kupferman, N. Piterman, and M. Y. Vardi, "Extended temporal logic revisited," in *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'01)*, ser. Lecture Notes in Computer Science, vol. 2154. Springer-Verlag, 2001, pp. 519–535.

[11] E. A. Emerson and E. M. Clarke, "Characterizing correctness properties of parallel programs using fixpoints," in *Proceedings of the 7th Colloquium on Automata, Languages and Programming (ICALP'80)*. Springer-Verlag, 1980, pp. 169–181.

[12] D. Kozen, "Results on the propositional μ-calculus," in *Proceedings of the 9th Colloquium on Automata, Languages and Programming (ICALP'82)*. Springer-Verlag, 1982, pp. 348–359.

[13] H. Barringer, R. Kuiper, and A. Pnueli, "A really abstract concurrent model and its temporal logic," in *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages (POPL'86)*, 1986, pp. 173–183.

[14] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh, "The temporal logic Sugar," in *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*. Springer-Verlag, 2001, pp. 363–367.

[15] M. Lange, "Weak automata for the linear time μ-calculus," in *Proceedings of the 6th Int'l. Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, ser. Lecture Notes in Computer Science, R. Cousot, Ed., vol. 3385. Sprinter-Verlag, 2005, pp. 267–281.

[16] M. Müller-Olm, "A modal fixpoint logic with chop," in *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS'99)*, ser. Lecture Notes in Computer Science, C. Meinel and S. Tison, Eds., vol. 1563. Springer-Verlag, 1999, pp. 510–520.

[17] D. Harel and D. Peleg, "Process logic with regular formulas," *Theoretical Computer Science*, vol. 38, pp. 307–322, 1985.

[18] J. G. Henriksen and P. S. Thiagarajan, "Dynamic linear time temporal logic," *Annals of Pure and Applied Logic*, vol. 96, no. 1–3, pp. 187–207, 1999.

[19] D. Fisman, C. Eisner, and J. Havlicek, *Formal syntax and Semantics of PSL: Appendix B of Accellera Property Language Reference Manual, Version 1.1*, March 2004.

[20] M. Lange, "Linear time logics around PSL: Complexity, expressiveness, and a little bit of succinctness," in *CONCUR*, ser. Lecture Notes in Computer Science, L. Caires and V. T. Vasconcelos, Eds., vol. 4703. Springer, 2007, pp. 90–104.

[21] C. Sánchez and M. Leucker, "Regular linear temporal logic with past," in *Proceedings of the 11th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'10)*, ser. Lecture Notes in Computer Science, G. Barthe and M. Hermenegildo, Eds., vol. 5944. Springer, 2010, pp. 295–311.

[22] M. Leucker and C. Sánchez, "Regular linear temporal logic," in *Proceedings of the 4th International Colloquium on Theoretical Aspects of Computing (ICTAC'07)*, ser. Lecture Notes in Computer Science, C. B. Jones, Z. Liu, and J. Woodcock, Eds., vol. 4711. Springer, 2007, pp. 291–305.

[23] L. J. Stockmeyer, "The computational complexity of word problems," Ph.D. dissertation, Massachusetts Institute of Technology, 1974.

[24] C. Dax and F. Klaedtke, "Alternation elimination by complementation," in *Proceedings of the 15th International Conference on Logic for Programming, Aritificial Intelligence and Reasoning (LPAR'08)*, ser. Lecture Notes in Computer Science, vol. 5530. Springer-Verlag, 2008, pp. 214–229.