Formalizing Quality of Reactive Systems

Shaull Almagor[1], Udi Boker[1,2], and Orna Kupferman[1]

[1] The Hebrew University, Jerusalem, Israel.
[2] IST Austria, Klosterneuburg, Austria.

Abstract. Temporal-logic model-checking is a successful paradigm for checking whether reactive systems satisfy specifications about their on-going behavior. The model-checking paradigm is Boolean, addressing the yes/no satisfiability question. We introduce and study a generalization of temporal logic and the model-checking paradigm, addressing a quality question. We define $\mathrm{LTL}^{\triangledown}$ – an extension of linear temporal logic with quality operators. The operators enable the specifier to prioritize different satisfying possibilities, to associate components of the specification with their criticality level, and to discount the influence of components on which we have low confidence. We develop a (weighted) automata-theoretic approach for reasoning about $\mathrm{LTL}^{\triangledown}$ specifications, solve its model-checking problem, and study its theoretical aspects.

1 Introduction

One of the main obstacles to the development of complex computerized systems lies in *verification* – the process of ensuring the systems' correctness. In *formal verification*, the system and its specification are mathematically modeled, and reasoning about the behavior of the system and its correctness is based on an algorithmic analysis of the possible computations of the system and the specification [9].

Formal-verification algorithms differ in the way systems and specifications are modeled. In *temporal-logic model checking*, a system is modeled by a labeled transition graph, and the specification is given by means of a temporal-logic formula. Temporal logic is derived from propositional logic by adding temporal operators, like G ("always"), F ("eventually"), X ("next"), and U ("until"), which describe a temporal ordering of events. For example, the temporal-logic formula $\mathsf{G}(req \rightarrow \mathsf{F}grant)$ specifies that every request is eventually granted. Linear temporal logic (LTL) has a fundamental role in the theory of formal verification and it is at the heart of industrial-strength specification formalisms [12, 29].

Model checking is Boolean. The Boolean fate of a verification process seems natural, as a system can either satisfy its specification or not satisfy it. The richness of today's systems, however, justifies specification formalisms that are *multi-valued*. The multi-valued setting arises directly in systems in which the designer can give to the atomic propositions rich values like "uninitialized", "unknown", "don't care", and more [19], and arises indirectly in applications like abstraction methods, query checking, and verification of systems from inconsistent viewpoints, where multi values are used to model missing, hidden, or varying information [5, 23]. Even broader than the examples above are settings in which the satisfaction values are taken from an unbounded domain, such as the set of real numbers. There, multi values are used for reasoning about *probabilistic systems* or about *quantitative properties* of systems. For example, it can be used in order to associate computations with the average waiting time between responses and grants, or with the energy consumption along

the computation. In recent years, the multi-valued setting has been an active area of research, offering promising results and tools for values taken from finite lattices [24], and in the context of probabilistic systems [11, 26] and quantitative verification [3, 7, 10, 17]. No attempts, however, have been made to augment temporal logics with a quantitative layer that would enable the specification of the relative merits of different aspects of the specification and would enable to formalize the *quality* of a reactive system.

In this paper we add a *quality measure* to the satisfiability of specifications of reactive systems, and we use it in order to formally define and reason about quality of systems. Our working assumption is that satisfying a specification is not a yes/no matter. Different ways of satisfying a specification should induce different levels of quality, which should be reflected in the output of the verification procedure. Consider for example the specification $\mathsf{G}(req \rightarrow \mathsf{F}grant)$. There should be a difference between a computation that satisfies it with grants generated soon after requests, one that satisfies it with long waits, one that satisfies it with several grants given to a single request, one that satisfies it vacuously (with no requests), and so on. Moreover, we may want to associate different levels of importance to different components of a specification, or we may have different levels of confidence about some of the components. We introduce and study $\mathrm{LTL}^{\triangledown}$: a multi-valued temporal logic that enables the specifier to associate specifications with a measure describing the quality in which they are satisfied.

Quality is a rather subjective issue. Technically, we can talk about the quality of satisfaction of specifications since there are different ways to satisfy specifications. Our formalism is based on new operators that enable the specifier to value these different ways. We distinguish between three orthogonal sources for quality (or lack of quality) and include in $\mathrm{LTL}^{\triangledown}$ the following three new *quality operators*:

- A *competence* operator $\triangledown_\lambda$, for $0 \leq \lambda \leq 1$. The satisfaction level of a specification $\triangledown_\lambda \psi$ is a value in $[0, 1]$, obtained by multiplying the satisfaction value of ψ by λ, where we associate `True` with 1 (full satisfaction) and `False` with 0 (no satisfaction). The more content we are with the way the specification is satisfied, the higher the satisfaction value is. For example, in the specification $\mathsf{G}(req \rightarrow (grant \vee \triangledown_{\frac{3}{4}} \mathsf{X} grant))$, we are fully happy only when all requests are immediately granted, and postponing a grant results in a smaller satisfaction value.
- A *necessity* operator $\blacktriangledown_\lambda$, for $0 \leq \lambda \leq 1$. The satisfaction level of a specification $\blacktriangledown_\lambda \psi$ is value in $[-1, 0]$, obtained by multiplying the satisfaction value of ψ by λ, where we associate `True` with 0 (no problems) and `False` with -1 (problems with critical components). The necessity operator addresses the fact that some components in the specification are a must while others are only nice to have. Consider for example a specification that states the system's "correctness" requirements φ, its "performance" requirements ψ, and its "appearance" requirements ξ. One may wish to consider the correctness requirements as a must, while putting partial necessity on the performance and appearance. This can be done with the necessity operator, for instance, by $\varphi \wedge \blacktriangledown_{\frac{1}{2}} \psi \wedge \blacktriangledown_{\frac{1}{4}} \xi$.
- A *confidence* operator Ⓥ_λ, for $0 \leq \lambda \leq 1$. The satisfaction value of a specification $\text{Ⓥ}_\lambda \psi$ is a value in $[-1, 1]$, obtained by multiplying the confidence value of ψ by λ, where we associate `True` with 1 (full confidence that ψ holds) and `False` with -1 (full confidence that ψ does not hold). The more confident we are about the way the specification is satisfied, the closer the satisfaction value is to 1 or -1, with satisfaction value 0 standing for no evidence neither for ψ's truth nor its falseness. For example, if a component ψ in the specification is an approximation of an accurate specification that

is too long for handling, or if ψ is evaluated on an abstraction of the system, we may want to reduce the influence of ψ's satisfaction, whether it is positive or negative.

The competence and the necessity operators are dual in the sense that while the competence operator reduces the truth level, the necessity operator reduces the falseness level. Then, the confidence operator reduces both the truth and falseness levels. Together, the three operators cover the possible linear approaches of formalizing quality as a value between true and false. For normalization purposes, one may evaluate all the three quality operators over the same range. The logic $\mathrm{LTL}^{\triangledown}$ indeed combines all operators, returning a quality measure in the range $[0, 1]$.

The model-checking problem for $\mathrm{LTL}^{\triangledown}$ can be stated as both a decision and a search problem. In the decision variant, we are given a system $\mathcal{K}$, an $\mathrm{LTL}^{\triangledown}$ formula φ, and a threshold $v \in [0, 1]$, and we have to decide whether all the computations of $\mathcal{K}$ satisfy φ with value at least v. In the search variant, we are looking for a maximal such threshold. As it turns out, it is possible to bound the number of values that a formula may be satisfied with, so the search variant of model checking can be reduced to the decision variant.

We describe two algorithms for solving the model-checking decision problem, both with a PSPACE complexity, as is Boolean LTL model checking [31]. The first algorithm is based on a reduction to Boolean LTL model checking. A key step in this algorithm is a transformation of $\mathrm{LTL}^{\triangledown}$ formulas to a normal form. In normal-formed formulas, negations and quality operators can be applied only on atomic propositions and literals, respectively. The transformation to the normal form is involved, and in particular it forces us to work with combined literals – ones that include additive constants, but it reveals helpful properties of specifications in $\mathrm{LTL}^{\triangledown}$, like monotonicity of quality. Using these properties, we can associate with each $\mathrm{LTL}^{\triangledown}$ formula φ and threshold v, an LTL formula $\mathrm{at\_least}(\varphi, v)$ such that a system satisfies $\mathrm{at\_least}(\varphi, v)$ iff the satisfaction value of φ in the system is at least v.

Our second model-checking algorithm is *automata based*. The automata-theoretic approach uses the theory of automata as a unifying paradigm for system specification, verification, and synthesis [33]. By viewing computations as words, we can view both the system and the specification as languages. Model checking then amounts to checking containment of the language of the system by the language of the specification. In the Boolean setting, the automata-theoretic approach has proven to be very useful and versatile [18, 20, 25, 32]. Traditional automata accept or reject their input, and are therefore Boolean. A *weighted automaton* maps each word to a value from a wide domain. The rich structure of weighted automata makes them intriguing mathematical objects. Fundamental problems that have been solved decades ago for non-weighted automata are still open or known to be undecidable in the weighted setting. In particular, the weighted variant of the language-containment problem, which is needed for solving the model-checking problem, is known to be generally undecidable [1, 21].

We describe a translation of $\mathrm{LTL}^{\triangledown}$ formulas to weighted automata, and use it in order to solve the model-checking problem for $\mathrm{LTL}^{\triangledown}$. In the Boolean setting, a simple translation of LTL to automata goes via alternating automata. The structure of LTL formulas guarantees that the automata are of a restricted type, which can be exploited by the model-checking algorithm [16]. We show that in the weighted setting, a similar simple translation exists. Moreover, the alternating automata we obtain are of a restricted type for which alternation can be removed. This is a key observation, as in general, unlike the Boolean setting, alternation cannot be removed in weighted automata, which is the source of several undecidability results.

Each of the two algorithms above has its own interesting benefits. The first, which reduces the problem to the Boolean setting, distills the theoretical differences between the two settings. Moreover, it allows to take advantage of known algorithms and tools. The second algorithm, using an automata-theoretic approach, provides a direct solution and a framework for further extensions of our setting.

Finally, we introduce and study two useful extensions of $\mathrm{LTL}^{\triangledown}$. The first extends the setting to weighted systems, and the second adds quality operators to branching-time temporal logics, namely to CTL*. We show that both extensions are easy to handle, and that their model-checking problems stay in PSPACE.

2 The Logic $\mathrm{LTL}^{\triangledown}$

In this section we introduce the temporal logic $\mathrm{LTL}^{\triangledown}$, which augments LTL by the ability to refer to the three quality measures described in Section 1. We first define three different logics, $\triangledown$-LTL, $\blacktriangledown$-LTL, and $\blacktriangledown\!\!\!\!\blacktriangledown$-LTL, corresponding to the three quality operators described there, and then unify them into one logic.

We start by defining the syntax. Let AP be a set of Boolean atomic propositions. For $\mathbb{V} \in \{\triangledown, \blacktriangledown, \blacktriangledown\!\!\!\!\blacktriangledown\}$, a $\mathbb{V}$-LTL formula is one of the following:

- $\mathtt{True}$, $\mathtt{False}$, or p, for $p \in AP$.
- $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\mathbb{V}_\lambda\varphi$, $\mathsf{X}\varphi$, $\varphi\mathsf{U}\psi$, or $\varphi\mathsf{R}\psi$, for $\mathbb{V}$-LTL formulas φ and ψ.

The semantics of the logics is defined with respect to (finite or infinite) computations over AP. We use $(2^{AP})^\infty$ to denote $(2^{AP})^* \cup (2^{AP})^\omega$. A computation is then a word $\pi = \pi_0, \pi_1, \ldots \in (2^{AP})^\infty$. We use π^i to denote the suffix $\pi_i, \pi_{i+1}, \ldots$. The semantics maps a computation π and a $\mathbb{V}$-LTL formula φ to the satisfaction value $[\![\pi, \varphi]\!]$. The range of the value depends on the logic: $[0,1]$, $[-1,0]$, and $[-1,1]$ for $\triangledown$-LTL, $\blacktriangledown$-LTL, and $\blacktriangledown\!\!\!\!\blacktriangledown$-LTL, respectively. The semantics is described in Table 1 below.[1]

Formula	$\triangledown$-LTL	$\blacktriangledown$-LTL	$\blacktriangledown\!\!\!\!\blacktriangledown$-LTL
$[\![\pi, \mathtt{True}]\!]$	1	0	1
$[\![\pi, \mathtt{False}]\!]$	0	-1	-1
$[\![\pi, \neg\varphi]\!]$	$1 - [\![\pi, \varphi]\!]$	$-(1 + [\![\pi, \varphi]\!])$	$-[\![\pi, \varphi]\!]$
$[\![\pi, p]\!]$	$[\![\pi, \mathtt{True}]\!]$ if $p \in \pi_0$ $[\![\pi, \mathtt{False}]\!]$ if $p \notin \pi_0$		
$[\![\pi, \varphi \vee \psi]\!]$	$\max([\![\pi, \varphi]\!], [\![\pi, \psi]\!])$		

Formula	$\triangledown$-LTL	$\blacktriangledown$-LTL	$\blacktriangledown\!\!\!\!\blacktriangledown$-LTL
$[\![\pi, \varphi \wedge \psi]\!]$	$\min([\![\pi, \varphi]\!], [\![\pi, \psi]\!])$		
$[\![\pi, \mathbb{V}_\lambda\varphi]\!]$	$\lambda \cdot [\![\pi, \varphi]\!]$		
$[\![\pi, \mathsf{X}\varphi]\!]$	$[\![\pi^1, \varphi]\!]$		
$[\![\pi, \varphi\mathsf{U}\psi]\!]$	$\max\limits_{0 \le i < \lvert\pi\rvert} [\min([\![\pi^i, \psi]\!], \min\limits_{0 \le j < i} [\![\pi^j, \varphi]\!])]$		
$[\![\pi, \varphi\mathsf{R}\psi]\!]$	$\min\limits_{0 \le i < \lvert\pi\rvert} [\max([\![\pi^i, \psi]\!], \max\limits_{0 \le j < i} [\![\pi^j, \varphi]\!])]$		

Table 1. The semantics of $\mathbb{V}$-LTL. The operator $\mathbb{V}$ stands for $\triangledown$, $\blacktriangledown$, and $\blacktriangledown\!\!\!\!\blacktriangledown$, according to the logic.

Note that Boolean LTL can be viewed as a special case of all the three logics. Indeed, without the quality operators, all formulas are mapped to $\mathtt{True}$ or $\mathtt{False}$.

2.1 A unified logic

As discussed in Section 1, the three quality operators cover different aspects of quality and its specification. It is useful to keep in mind the fact that while the competence operator

[1] The observant reader may be concerned by our use of $\max$ and $\min$ where $\sup$ and $\inf$ are in order. In Lemma 1 we prove that there are only finitely many satisfaction values for a formula φ, and thus the semantics is well defined.

reduces the truth level, the necessity operator reduces the falseness level. Then, the confidence operator reduces both the truth and falseness levels. Formally, for all $0 < \lambda < 1$, we have that $\triangledown_\lambda \texttt{True} \neq \texttt{True}$ and $\triangledown_\lambda \texttt{False} = \texttt{False}$, whereas $\blacktriangledown_\lambda \texttt{True} = \texttt{True}$ and $\blacktriangledown_\lambda \texttt{False} \neq \texttt{False}$. Also, $\blacktriangledown\!\!\!\!\triangledown_\lambda \texttt{True} \neq \texttt{True}$ and $\blacktriangledown\!\!\!\!\triangledown_\lambda \texttt{False} \neq \texttt{False}$.

We now wish to unify our three logics into one that enables the expression of the three types of quality measures. For this, we first unify the ranges of the satisfaction values, and then adjust the semantics to the new range. We (arbitrarily) choose to work over $[0, 1]$.

The $\blacktriangledown$ operator over $[0, 1]$ and via $\triangledown$. The natural linear mapping $\mu : [-1, 0] \to [0, 1]$ is defined by $\mu(x) = x + 1$, and accordingly $\mu^{-1}(y) = y - 1$. We define the semantics of $\blacktriangledown$ over $[0, 1]$ by mapping a $[0, 1]$-value to $[-1, 0]$, applying the semantics of $\blacktriangledown$-LTL, and mapping the result back to $[0, 1]$. It turns out that the semantics of all the operators is identical to their semantics in $\triangledown$-LTL, except for the $\blacktriangledown_\lambda$ operator, for which we have that $[\![\pi, \blacktriangledown_\lambda \varphi]\!] = \mu(\blacktriangledown_\lambda \mu^{-1}([\![\pi, \varphi]\!])) = \mu(\lambda([\![\pi, \varphi]\!] - 1)) = \lambda[\![\pi, \varphi]\!] + (1 - \lambda)$. In terms of expressing $\blacktriangledown$ with $\triangledown$, it is easy to verify that $[\![\pi, \blacktriangledown_\lambda \varphi]\!] = [\![\pi, \neg\triangledown_\lambda \neg\varphi]\!]$.

The $\blacktriangledown\!\!\!\!\triangledown$ operator over $[0, 1]$ and via $\triangledown$. The natural linear mapping $\eta : [-1, 1] \to [0, 1]$ is defined by $\eta(x) = (x + 1)/2$, and accordingly $\eta^{-1}(y) = 2y - 1$. As in the case of the necessity operator, it turns out that the semantics of all operators is identical to their semantics in $\triangledown$-LTL, except for the $\blacktriangledown\!\!\!\!\triangledown_\lambda$ operator, for which we have that $[\![\pi, \blacktriangledown\!\!\!\!\triangledown_\lambda \varphi]\!] = \eta(\blacktriangledown\!\!\!\!\triangledown_\lambda(\eta^{-1}([\![\pi, \varphi]\!]))) = \eta(\blacktriangledown\!\!\!\!\triangledown_\lambda(2[\![\pi, \varphi]\!] - 1)) = \eta(\lambda(2[\![\pi, \varphi]\!] - 1)) = (\lambda(2[\![\pi, \varphi]\!] - 1) + 1)/2 = \lambda[\![\pi, \varphi]\!] + (1 - \lambda)/2$. In terms of expressing $\blacktriangledown\!\!\!\!\triangledown$ by $\triangledown$, we get that $[\![\pi, \blacktriangledown\!\!\!\!\triangledown_\lambda \varphi]\!] = [\![\pi, \triangledown_\lambda \varphi]\!] + (1 - \lambda)/2$.

As we can see, it is possible to express the $\blacktriangledown$ and $\blacktriangledown\!\!\!\!\triangledown$ operators using the $\triangledown$ operator, yet for the latter the translation requires additive constants, which are not part of the syntax of $\triangledown$-LTL. As it turns out, however, the additive constants can be easily handled by the algorithms we are going to use. Thus, we are happy with the above translation, and conclude by explicitly stating the semantics of the $\blacktriangledown$ and $\blacktriangledown\!\!\!\!\triangledown$ operators in the unified logic (for all other operators, the semantics is identical to the one of $\triangledown$-LTL given in Table 1).

- $[\![\pi, \blacktriangledown_\lambda \varphi]\!] = \lambda[\![\pi, \varphi]\!] + (1 - \lambda)$.
- $[\![\pi, \blacktriangledown\!\!\!\!\triangledown_\lambda \varphi]\!] = \lambda[\![\pi, \varphi]\!] + (1 - \lambda)/2$.

Finally, we extend the semantics of $\mathrm{LTL}^\triangledown$ to Kripke structures. A *Kripke structure* is a tuple $\mathcal{K} = \langle AP, S, I, \rho, L\rangle$, where AP is a finite set of atomic propositions, S is a finite set of states, $I \subseteq S$ is the set of initial states, $\rho \subseteq S \times S$ is a total transition relation, and $L : S \to 2^{AP}$ is a labeling function. A *trace* of $\mathcal{K}$ is a (finite or infinite) sequence $s = s_0, s_1, \ldots$ of states such that $s_0 \in I$ and for every $0 \leq i < |s|$, it holds that $\langle s_i, s_{i+1}\rangle \in \rho$. A word $\pi = \pi_0, \pi_1, \ldots$ over 2^{AP} is a *computation* of $\mathcal{K}$ if there exists a trace s of $\mathcal{K}$ such that $\pi_i = L(s_i)$ for all $0 \leq i < |\pi|$, and $|s| = |\pi|$.

In the Boolean setting of LTL, a Kripke structure satisfies a formula φ if all its computations satisfy the formula. Adopting this universal approach, the satisfaction value of an $\mathrm{LTL}^\triangledown$ formula φ in a Kripke structure $\mathcal{K}$, denoted $[\![\mathcal{K}, \varphi]\!]$, is induced by the "worst" computation of $\mathcal{K}$, namely the one in which φ has the minimal satisfaction value. Formally, $[\![\mathcal{K}, \varphi]\!] = \min\{[\![\pi, \varphi]\!] : \pi \text{ is a computation of } \mathcal{K}\}$. [2]

[2] Since a Kripke structure may have infinitely many computations, we should a-priori use $\inf$ rather than $\min$. As we prove, however, in Lemma 1, only finitely many values are possible, thus our semantics is well defined.

Example 1. Consider the following specification to a disc on key:

$$\mathsf{G}(\mathit{buffer\_full} \rightarrow \mathsf{X}((\mathit{write}_1 \wedge \mathit{write}_2) \vee \triangledown_{\frac{3}{4}}\, \mathit{write}_1) \wedge (\blacktriangledown_{\frac{1}{3}}\, \mathit{light\_on}) \wedge \blacktriangledown_{\frac{5}{6}}(\neg \mathit{abort})\mathsf{U}\mathit{buffer\_empty}).$$

The specification states that the designer prefers that both write_1 and write_2 are on after the buffer is full, the light has to be on, but this is not a critical requirement, and the writing operation should not abort until completion. Since the specification abstracts possible reasons for a justified abort, the confidence in the latter requirement is not full.

In Appendix A.1 we demonstrate the usefulness of $\mathrm{LTL}^{\triangledown}$ with more examples. In particular, we show there how $\mathrm{LTL}^{\triangledown}$ can be used in order to add a quantitative layer to the notions of vacuity and coverage [22].

2.2 A normal form for $\mathrm{LTL}^{\triangledown}$

We seek a normal form for $\mathrm{LTL}^{\triangledown}$ formulas, in which the only quality operator is $\triangledown$, and negations and $\triangledown$ can be applied only on atomic propositions and literals, respectively. We have seen that expressing $\blacktriangledown$ and $\blacktriangledown$ with $\triangledown$ is possible, but requires the extension of the logic by a constant-addition operator. Such an extension is needed also in the process of pushing negations inwards through $\triangledown$. Indeed, already for atomic propositions, we have that

$$[\![\pi, \neg\triangledown_\lambda p]\!] = \begin{cases} 1-\lambda & \text{if } p \in \pi_0 \\ 1 & \text{if } p \notin \pi_0, \end{cases} \quad \text{whereas} \quad [\![\pi, \triangledown_\lambda \neg p]\!] = \begin{cases} 0 & \text{if } p \in \pi_0 \\ \lambda & \text{if } p \notin \pi_0. \end{cases}$$

Still, for every $\mathrm{LTL}^{\triangledown}$ formula φ, we have that $[\![\pi, \neg\triangledown_\lambda \varphi]\!] = 1 - \lambda [\![\pi, \varphi]\!] = (1-\lambda) + \lambda - \lambda [\![\pi, \varphi]\!] = (1-\lambda) + [\![\pi, \triangledown_\lambda \neg\varphi]\!]$. Thus, we introduce a *constant-addition operator*, $+\kappa$, for $\kappa \in [-1, 1]$ (we need negative values since expressing $\blacktriangledown$ with $\triangledown$ may require subtracting a value), with the semantics $[\![\pi, \varphi + \kappa]\!] = [\![\pi, \varphi]\!] + \kappa$. The constant-addition operator allows us to use the quasi-commutativity of $\neg$ and $\triangledown$ to obtain a normal form. The normal form is going to be useful in algorithms for $\mathrm{LTL}^{\triangledown}$, and, as we discuss in the sequel, it nicely explains some of its theoretical properties.

Formally, an $\mathrm{LTL}^{\triangledown}$ formula in normal form is one of the following:

- `True`, `False`, $\triangledown_\lambda p + \kappa$, or $\triangledown_\lambda \neg p + \kappa$, for $p \in AP$, $\kappa \in [-1, 1]$, and $\lambda \in (0, 1)$.
- $\varphi \vee \psi$, $\varphi \wedge \psi$, $\mathsf{X}\varphi$, $\varphi \mathsf{U} \psi$, or $\varphi \mathsf{R} \psi$, for $\mathrm{LTL}^{\triangledown}$ formulas φ and ψ in normal form.

We refer to formulas of the form $\triangledown p + \kappa$ or $\triangledown \neg p + \kappa$ as *combined literals*. Note that in a formula in normal form, quality operators are applied only in the propositional level. Still, as we now show, the accumulation of values in combined literals covers for this:

Theorem 1. *Every* $\mathrm{LTL}^{\triangledown}$ *formula has an equivalent formula of linear length in* $\mathrm{LTL}^{\triangledown}$ *normal form.*

It is important to note that general accumulation of the additive constants in the combined literals may result in a value that is not in $[0, 1]$. This is, however, not the case in formulas that are obtained by the translation in Theorem 1.

Finally, we prove the following simple yet important lemma, which justifies the use of $\max$ and $\min$ (rather than sup and inf) in the semantics. This lemma is also handy in the formulation of the model-checking problem as a as a decision rather than a search problem.

Lemma 1. *Consider an* $\mathrm{LTL}^{\triangledown}$ *formula* φ *in normal form, over the combined literals* $\kappa_1 + \triangledown_{\lambda_1} l_1, \ldots, \kappa_n + \triangledown_{\lambda_n} l_n$. *The satisfaction value of* φ *in every computation is in* $\{0, 1, \kappa_1 + \lambda_1, \ldots, \kappa_n + \lambda_n, \kappa_1, \ldots, \kappa_n\}$.

3 LTL$^\nabla$ Model Checking

The *Model-checking* (decision) problem for LTL$^\nabla$ is to decide, given a Kripke structure $\mathcal{K}$, an LTL$^\nabla$ formula φ, and a threshold $v \in [0,1]$, whether $[\![\mathcal{K},\varphi]\!] \geq v$. A search version of this problem is to compute the maximal v for which $[\![\mathcal{K},\varphi]\!] \geq v$ holds. By Lemma 1, the search query can be solved by solving a logarithmic number of decision queries. We thus focus on the decision problem. Also, per Theorem 1, we assume that LTL$^\nabla$ formulas are given in normal form.

In this section we present a solution to the model-checking problem. The solution is based on a reduction to the model-checking problem for LTL, and it applies to both finite and infinite computations. The reduction is based on the observation that LTL$^\nabla$ formulas in normal form are *monotonic*: increasing the satisfaction value of a subformula (e.g., by replacing it with `True`) of an LTL$^\nabla$ formula φ in normal form can only increase the satisfaction value of φ. Dually, decreasing the value of a subformula (e.g., by replacing it with `False`) can only decrease the satisfaction value of φ.

For a combined literal $\theta = \kappa + \nabla_\lambda l$ and a value $c \in [0,1]$, we define a Boolean propositional formula $\mathrm{at\_least}(\theta, c)$ such that a Boolean assignment $\sigma \subseteq AP$ to the atomic propositions satisfies $\mathrm{at\_least}(\theta, c)$ iff the satisfaction value of θ with respect to σ is at least c. We define $\mathrm{at\_least}(\theta, c)$ as follows. If $c \leq \kappa$ then $\mathrm{at\_least}(\theta, c) = \texttt{True}$, if $\kappa < c \leq \kappa + \lambda$ then $\mathrm{at\_least}(\theta, c) = l$, and if $\kappa + \lambda < c$ then $\mathrm{at\_least}(\theta, c) = \texttt{False}$. We extend the definition to LTL$^\nabla$ formulas. Thus, for an LTL$^\nabla$ formula φ and a value $c \in [0,1]$, we define $\mathrm{at\_least}(\varphi, c)$ to be the LTL formula obtained from φ by replacing every combined literal θ by $\mathrm{at\_least}(\theta, c)$.

Lemma 2. *Consider a computation π, an* LTL$^\nabla$ *formula φ, and a threshold $v \in [0,1]$. Then, $[\![\pi,\varphi]\!] \geq v$ iff π satisfies* $\mathrm{at\_least}(\varphi, v)$.

We can now conclude with the following.

Theorem 2. *The model-checking problem for* LTL$^\nabla$ *is PSPACE-complete.*

Proof: We start with decision queries. Consider a Kripke structure $\mathcal{K}$, an LTL$^\nabla$ formula φ in normal form, and a threshold $v \in [0,1]$. By Theorem 2, for every computation π of $\mathcal{K}$, we have that $[\![\pi,\varphi]\!] \geq v$ iff π satisfies $\mathrm{at\_least}(\varphi, v)$. Recall that $[\![\mathcal{K},\varphi]\!] = \min\{[\![\pi,\varphi]\!] : \pi \text{ is a computation of } \mathcal{K}\}$. Accordingly, $[\![\mathcal{K},\varphi]\!] \geq v$ iff π satisfies $\mathrm{at\_least}(\varphi, v)$ for all the computations π of $\mathcal{K}$, which holds iff $\mathcal{K}$ satisfies $\mathrm{at\_least}(\varphi, v)$. By [31, 33], the latter can be checked in nondeterministic space that is logarithmic in $\mathcal{K}$ and polynomial in φ, thus the problem is in PSPACE. Hardness in PSPACE follows form hardness of LTL model checking.

For search queries, given $\mathcal{K}$ and φ, there are, per Lemma 1, $O(|\varphi|)$ possible values for $[\![\mathcal{K},\varphi]\!]$. Thus, model checking can be reduced to a sequence of $O(\log|\varphi|)$ decision queries, and is PSPACE-complete. □

4 An Automata-Theoretic Approach

In this section we describe a translation of LTL$^\nabla$ formulas to weighted automata, and use it in order to solve the model-checking problem for LTL$^\nabla$. As discussed in Section 1, the automata we use are weighted alternating automata of a restricted type, resembling the very weak alternating automata used in [16] for Boolean LTL model checking. We first introduce

the weighted automata model that we use, and then describe the model-checking procedure that is based on it. The complexity of the algorithm coincides with the one presented in Section 3. The automata-theoretic approach, however, is more versatile, providing a framework for future extensions, such as discounting of the quality operators along a computation.

Alternating weighted automata. We start by defining our variant of weighted alternating automata. Below we give the definition in brief. Full details can be found in Appendix A.5.

For a given set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$). For $Y \subseteq X$, we say that Y *satisfies* a formula $\theta \in \mathcal{B}^+(X)$ iff the truth assignment that assigns *true* to the members of Y and assigns *false* to the members of $X \setminus Y$ satisfies θ. A *weighted alternating automaton* is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$, where Σ is a finite non-empty alphabet, Q is a finite non-empty set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma \to \mathcal{B}^+(Q \times \mathbb{Q})$ is a weighted transition function, and $\alpha \subseteq Q$ is a set of accepting states. A run of the automaton starts in q_0. When the automaton is in state q and it reads a letter σ, it has to satisfy all the weighted requirements specified in $\delta(q, \sigma)$. If, for example, the input word is $\sigma \cdot w$ and $\delta(q_0, \sigma) = \langle q_1, 2 \rangle \wedge (\langle q_2, 4 \rangle \vee \langle q_3, 1 \rangle)$, then the automaton has two choices: one is to continue and read w from both q_1 and q_2, and the second is to read w from both q_1 and q_3. The transition from q_0 to q_1 has weight 2, the one to q_2 has weight 4, and the one to q_3 has weight 1.

Accordingly, a run of a weighted alternating automaton on a word $w = w_0, w_1, w_2, \ldots$ is a weighted tree. The root of the tree is labeled by q_0. The successors of a node in level l that is labeled by state q correspond to the choice the automaton has made in satisfying $\delta(q, w_l)$. In the example above, the run tree has a root (one node in level 0) labeled q_0, and has two nodes in level 1. In the first choice, the nodes are labeled q_1 and q_2, and in the second they are labeled q_1 and q_3. The edges of the tree are labeled by the corresponding weights. The run then continues in a similar way from the nodes in level 1. A run is accepting if it satisfies the acceptance conditions. In the case of finite words, all the leaves in the run tree should be in α. In the case of infinite words, we work with the Büchi acceptance condition and all the infinite paths in the run tree should visit α infinitely often.

There are several possible ways to refer to the weights in the run tree. For example, in weighted automata defined with respect to the tropical semiring [28], one sums the weights along finite paths, and in the alternating weighted automata of [8], which run on infinite words, one takes their sup or their limit average. In our alternating automata, the value of a path in the run tree is the product of all the weights along it. Our semantics interpret weights as *rewards* – something that the automaton wants to maximize. Accordingly, nondeterminism ($\vee$ in the transitions) is interpreted as maximum: the value of a word is the value of the best (heaviest) run, and universality ($\wedge$ in the transitions) is interpreted as minimum: the value of a run is the minimal weight of the path in it. Note that a weighted automaton $\mathcal{A}$ can be viewed as a partial function from Σ^∞ to $\mathbb{Q}$. (The function is partial since some words need not have an accepting run). We use $\mathcal{A}(w)$ to denote the value that $\mathcal{A}$ assigns to a word w.

An alternating automaton is *nondeterministic* if the only operator in the formulas in the transition function is $\vee$. Observe that in a nondeterministic automaton, a run is a path (that is, the branching degree of every run tree is 1). Thus, the weight of a run is simply the product of the weights along the run. We use WAFW and WNFW to denote weighted alternating and nondeterministic automata on finite words, respectively, and use WABW and WNBW to denote the Büchi automata counterparts.

Note that since the value of a run is based on product, the value is indifferent to transitions with weight 1. We say that a weighted alternating automaton $\mathcal{A}$ is *very weak* if the only cycles in $\mathcal{A}$ are self loops with weight 1 and every path in a run of $\mathcal{A}$ has at most one edge whose weight is not 1 (the name is analogous to very weak alternating automata in the Boolean case [16]). Finally, we say that a weighted nondeterministic automaton $\mathcal{A}$ is *simple* if all its cycles have weight 1.

From $\mathrm{LTL}^{\triangledown}$ *to weighted automata.* We can now present a translation of $\mathrm{LTL}^{\triangledown}$ formulas to weighted automata. The general scheme is similar to the Boolean setting: given an $\mathrm{LTL}^{\triangledown}$ formula φ, we translate it into an alternating weighted automaton $\mathcal{A}_\varphi$ such that for every computation π, it holds that $[\![\pi, \varphi]\!] = \mathcal{A}_\varphi(\pi)$. The construction coincides for finite and infinite words, except that in the first we view the result as an automaton on finite words and in the second as a Büchi automaton.

Theorem 3. *Consider an* $\mathrm{LTL}^{\triangledown}$ *formula* φ*. There is a very weak weighted alternating automaton* $\mathcal{A}_\varphi$ *such that for every computation* $\pi \in (2^{AP})^\infty$*, it holds that* $[\![\pi, \varphi]\!] = \mathcal{A}_\varphi(\pi)$*, and the size of* $\mathcal{A}_\varphi$ *is* $O(|\varphi|)$*.*

Proof: For an $\mathrm{LTL}^{\triangledown}$ formula φ in normal form, we define the *closure* of φ, denoted $cl(\varphi)$, as the set of all subformulas of φ. Given φ, we define $\mathcal{A}_\varphi = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$, where $\Sigma = 2^{AP}$, $Q = cl(\varphi) \cup \{\top\}$, $q_0 = \varphi$, and δ and α are defined as follows. First, for every letter $\sigma \in 2^{AP}$, the transition function is as follow.

- $\delta(\kappa + \triangledown_\lambda p, \sigma) = \begin{cases} \langle \top, \kappa + \lambda \rangle & \text{if } p \in \sigma, \\ \langle \top, \kappa \rangle & \text{if } p \notin \sigma. \end{cases}$
- $\delta(\kappa + \triangledown_\lambda \neg p, \sigma) = \begin{cases} \langle \top, \kappa + \lambda \rangle & \text{if } p \notin \sigma, \\ \langle \top, \kappa \rangle & \text{if } p \in \sigma. \end{cases}$
- $\delta(\psi_1 \mathsf{U} \psi_2, \sigma) = \delta(\psi_2, \sigma) \vee (\delta(\psi_1, \sigma) \wedge \langle \psi_1 \mathsf{U} \psi_2, 1 \rangle)$.
- $\delta(\psi_1 \mathsf{R} \psi_2, \sigma) = \delta(\psi_2, \sigma) \wedge (\delta(\psi_1, \sigma) \vee \langle \psi_1 \mathsf{R} \psi_2, 1 \rangle)$.
- $\delta(\top, \sigma) = \langle \top, 1 \rangle$.
- $\delta(\psi_1 \wedge \psi_2, \sigma) = \delta(\psi_1, \sigma) \wedge \delta(\psi_2, \sigma)$.
- $\delta(\psi_1 \vee \psi_2, \sigma) = \delta(\psi_1, \sigma) \vee \delta(\psi_2, \sigma)$.

The set of accepting states is $\alpha = \{\top\} \cup \{\psi_1 \mathsf{R} \psi_2 : \psi_1 \mathsf{R} \psi_2 \in cl(\varphi)\}$.

In Appendix A.6, we prove the correctness of the construction. Intuitively, when $\mathcal{A}_\varphi$ is in state ψ when it reads a computation ρ (in particular, a suffix of π), it follows the semantics of ψ in order to calculate $[\![\rho, \psi]\!]$. The analogy between min and max in the semantics and $\wedge$ and $\vee$ in the transitions of the automaton leads to a proof by an induction on the structure of ψ. Note that thanks to the normal form, quality operators appear only in transitions in which $\mathcal{A}_\varphi$ moves to the sink $\top$. This both simplifies the proof and implies that $\mathcal{A}_\varphi$ is very weak. □

Alternation removal. Alternating weighted automata are in general strictly more expressive than nondeterministic ones (see Appendix A.10). Recall that in very weak automata, every path in a run tree can have at most one edge whose weight is not 1. Accordingly, the value of a run is simply the minimal weight of an edge appearing in the run, which enables alternation removal. Moreover, the obtained nondeterministic automaton is simple. Intuitively, the construction only has to augment the subset construction by a component that maintains the minimal edge encountered so far.

Theorem 4. *Given a very weak WAFW (WABW) with* n *states, we can construct an equivalent simple WNFW (WNBW, respectively) with* $2^{O(n)}$ *states.*

By combining Theorems 3 and 4, we can conclude with the following.

Corollary 1. *Consider an* $\mathrm{LTL}^{\triangledown}$ *formula* φ*. There is a simple weighted nondeterministic automaton* $\mathcal{A}_\varphi$ *such that for every computation* $\pi \in (2^{AP})^\infty$*, it holds that* $[\![\pi, \varphi]\!] = \mathcal{A}_\varphi(\pi)$*, and the size of* $\mathcal{A}_\varphi$ *is* $2^{O(|\varphi|)}$*.*

Automata-based model checking. Consider a Kripke structure $\mathcal{K}$, an $\mathrm{LTL}^{\triangledown}$ formula φ, and a threshold $v \in [0,1]$. Note that $[\![K, \varphi]\!] \geq v$ iff there does not exist a computation π in $\mathcal{K}$ such that $[\![\pi, \neg\varphi]\!] > 1 - v$.

As detailed in Appendix A.8, we can define the product of $\mathcal{K}$ and a simple weighted nondeterministic automaton $\mathcal{A}$ as a simple weighted nondeterministic automaton $\mathcal{P}$ such that $\mathcal{P}(w)$ is defined if both w is a computation of $\mathcal{K}$ and $\mathcal{A}(w)$ is defined. In this case, $\mathcal{P}(w) = \mathcal{A}(w)$. The size of $\mathcal{P}$ is the product of the sizes of $\mathcal{K}$ and $\mathcal{A}$. Having constructed a weighted automaton $\mathcal{A}_{\neg\varphi}$ for (the normal form of) $\neg\varphi$, we can reduce the model-checking problem for $\mathcal{K}$, φ, and v, to the problem of deciding whether the product $\mathcal{P}$ of $\mathcal{K}$ and $\mathcal{A}_{\neg\varphi}$ accepts some word with value greater than $1 - v$. As detailed in Appendix A.9, this problem can be solved in PSPACE in the size of φ and v, and in NLOGSPACE in the size of $\mathcal{K}$, suggesting an alternative proof to Theorem 2.

5 Extensions

In this section we introduce and study two useful extensions of $\mathrm{LTL}^{\triangledown}$, handling weighted systems and branching-time temporal logics.

5.1 Weighted systems

A *weighted Kripke structure* is a tuple $\mathcal{K} = \langle AP, S, I, \rho, L\rangle$, where AP, S, I, and ρ are as in Boolean Kripke structures, and $L : S \to [0,1]^{AP}$ maps each state to a weighted assignment to the atomic propositions. Thus, the value $L(s)(p)$ of an atomic proposition $p \in AP$ in a state $s \in S$ is a value in $[0,1]$. The semantics of $\mathrm{LTL}^{\triangledown}$ with respect to a weighted computation coincides with the one for non-weighted systems, except that for an atomic proposition p, we have that $[\![\pi, p]\!] = L(\pi_0)(p)$.

It is not hard to extend the model-checking algorithm for $\mathrm{LTL}^{\triangledown}$ to the settings of weighted systems. Essentially, the extension of the algorithm is based on the fact that the $\mathrm{at\_least}(\theta, c)$ function defined in Section 3, can be adjusted to the weighted setting. Unlike the algorithm there, the reduction to Boolean LTL generates formulas over a fresh set of atomic propositions – ones that correspond to $\mathrm{at\_least}(\theta, c)$, but other than that, the algorithm is identical.

5.2 Adding a quality layer to branching temporal logics

Formulas of $\mathrm{LTL}^{\triangledown}$ specify on-going behaviors of linear computations. A Kripke structure is not linear, and the way we interpret $\mathrm{LTL}^{\triangledown}$ formulas with respect to it is universal. In *branching temporal logic* one can add universal and existential quantifiers to the syntax of the logic, and specifications can refer to the branching nature of the system [13]. The branching temporal logic $\mathrm{CTL}^{\star\triangledown}$ extends $\mathrm{LTL}^{\triangledown}$ by the path quantifiers E and A. Formulas of the form $\mathsf{E}\varphi$ and $\mathsf{A}\varphi$ are referred to as *state formulas* and they are interpreted over states s in the structure with the semantics $[\![s, \mathsf{E}\varphi]\!] = \max\{[\![\pi, \varphi]\!] \mid \pi \text{ starts in } s\}$ and $[\![s, \mathsf{A}\varphi]\!] = \min\{[\![\pi, \varphi]\!] \mid \pi \text{ starts in } s\}$.

In [14], the authors describe a general technique for extending the scope of LTL model-checking algorithms to CTL*. The idea is to repeatedly consider an innermost state subformula, view it as an (existentially or universally quantified) LTL formula, apply LTL model checking in order to evaluate it in all states, and add a fresh atomic proposition that replaces this subformula and holds in exactly these states that satisfy it. A naive attempt to use this technique in order to model check $\mathrm{CTL}^{\star\nabla}$ fails, as the satisfaction value of LTL^∇ formulas is not Boolean, and hence they cannot be replaced by atomic propositions. Fortunately, having solved the LTL^∇ model-checking problem for weighted systems, we can still use a variant of this technique: rather than replacing formulas by Boolean atomic propositions, replace them by weighted ones, corresponding to the solution of the search variant of LTL^∇ model checking. Hence, the model-checking problem for $\mathrm{CTL}^{\star\nabla}$ is PSPACE-complete, as are the ones for LTL^∇ and for CTL*. A direction for future research is to study fragments of $\mathrm{CTL}^{\star\nabla}$ for which model checking is simpler. In particular, our initial results show that for CTL^∇, in which each temporal operator must be preceded by a path quantifier, it is possible to apply a variant of the linear-time fixed-point based model-checking algorithm of CTL [15].

6 Directions for Future Research

We described a temporal logic that makes it possible to formalize the quality of reactive systems and studied its model-checking problem. We are now in the process of studying two orthogonal extensions of our work, which we find both interesting and important.

The first concerns the *expressive power* of LTL^∇ and its extension with the following two features. The first is temporal operators in which the future is discounted. This captures the intuition that events that happen close to the present affect the quality more than ones that happen in the far future. The technical challenge in the discounted extension is that the set of possible values that a formula may be assigned to is infinite. We still believe that the automata-theoretic approach presented here can handle this setting. The second feature is the ability to take the weighted average of different events. Here too, the added expressive power has a computational price, and naturally, there is the possibility of combining both features.

The second extension concerns *synthesis* of LTL^∇ formulas. In conventional synthesis algorithms we are given a specification to a reactive system, typically by means of an LTL formula, and we transform it into a system that is guaranteed to satisfy the specification with respect to all environments [30]. Very little attention has been payed to the quality of the systems that are automatically synthesized. Current efforts to address the "quality weakness" are based on enriching the game that corresponds to synthesis to a weighted one [2, 6]. Using LTL^∇, one will be able to have the weights and the quality specification embodied in the specification.

References

1. S. Almagor, U. Boker, and O. Kupferman. What's decidable about weighted automata? In *Proc. 9th ATVA, LNCS 6996*, pages 482–491, 2011.
2. R. Bloem, K. Chatterjee, T. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *Proc. 21st CAV, LNCS 5643*, pages 140–156, 2009.
3. U. Boker, K. Chatterjee, T.A. Henzinger, and O. Kupferman. Temporal specifications with accumulative values. In *Proc. 26th IEEE LICS*, pages 43–52, 2011.

4. U. Boker, O. Kupferman, and A. Rosenberg. Alternation removal in Büchi automata. In *Proc. 37th ICALP, LNCS 6199*, pages 76–87, 2010.
5. G. Bruns and P. Godefroid. Model checking with multi-valued logics. In *31st ICALP, LNCS 3142*, pages 281–293, 2004.
6. P. Cerný, K. Chatterjee, T.A. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In *23rd CAV*, pages 243–259, 2011.
7. K. Chatterjee, L. de Alfaro, M. Faella, T.A. Henzinger, R. Majumdar, and M. Stoelinga. Compositional quantitative reasoning. In *QEST*, pages 179–188, 2006.
8. K. Chatterjee, L. Doyen, and T. Henzinger. Alternating weighted automata. In *Proc. 17th FCT, LNCS 5699*, pages 3–13, 2009.
9. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. L. de Alfaro, M. Faella, and M. Stoelinga. Linear and branching metrics for quantitative transition systems. In *31st ICALP*, pages 97–109, 2004.
11. J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Metrics for labelled markov processes. *Theoretical Computer Science*, 318(3):323–354, 2004.
12. C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.
13. E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.
14. E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 84–96, 1985.
15. E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st IEEE LICS*, pages 267–278, 1986.
16. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proc. 13th CAV, LNCS 2012*, pages 53–65, 2001.
17. T.A. Henzinger. From Boolean to quantitative notions of correctness. In *Proc. 37th ACM Symp. on Principles of Programming Languages*, pages 157–158, 2010.
18. G.J. Holzmann. The model checker SPIN. *IEEE TSC*, 23(5):279–295, 1997.
19. IEEE. IEEE standard multivalue logic system for VHDL model interoperability, 1993.
20. B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Proc. 19th CAV, LNCS 4590*, pages 258–262, 2007.
21. D. Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. *International Journal of Algebra and Computation*, 4(3):405–425, 1994.
22. O. Kupferman. Sanity checks in formal verification. In *Proc. 17th CONCUR, LNCS 4137*, pages 37–51, 2006.
23. O. Kupferman and Y. Lustig. Lattice automata. In *Proc. 8th VMCAI, LNCS 4349*, pages 199 – 213, 2007.
24. O. Kupferman and Y. Lustig. Latticed simulation relations and games. *International Journal on the Foundations of Computer Science*, 21(2):167–189, 2010.
25. R.P. Kurshan. *FormalCheck User's Manual*. Cadence Design, Inc., 1998.
26. M.Z. Kwiatkowska. Quantitative verification: models techniques and tools. In *ESEC/SIGSOFT FSE*, pages 449–458, 2007.
27. S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *TCS*, 32:321–330, 1984.
28. M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.
29. A. Pnueli. The temporal semantics of concurrent programs. *TCS*, 13:45–60, 1981.
30. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 179–190, 1989.
31. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.
32. M.Y. Vardi. Automata-theoretic model checking revisited. In *8th VMCAI Conf.*, volume 4349 of *LNCS*, pages 137–150. Springer, 2007.
33. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *I&C*, 115(1):1–37, 1994.

A Appendix

A.1 Examples of useful LTL$^{\triangledown}$ formulas

Vacuity and coverage. Vacuity and coverage are two popular *sanity checks* in formal verification. They are applied after a successful verification process, aiming to check that all the components of the specification and the system have played a role in the satisfaction. Algorithms for measuring vacuity and coverage are based on model checking mutations of the system or the specification. For example, after a system is proven to satisfy the specification $\mathsf{G}(req \rightarrow \mathsf{F}grant)$, vacuity-detection algorithms would mutate the specification to $\mathsf{G}(req \rightarrow \texttt{False})$ or to $\mathsf{G}(\texttt{True} \rightarrow \mathsf{F}grant)$ in order to check that all components of the specification affect the satisfaction. Coverage-measuring algorithms would then mutate the system, say by removing some grants, in order to check whether all components of the system affect the satisfaction [22]. A serious drawback of vacuity and coverage lies in the fact that it is difficult to distinguish between different cases of vacuous satisfaction and low coverage. While some cases should alarm the designer, some are a natural part of the design. In the example above, it is more alarming to discover that the specification has been satisfied in a system where no requests are issued, than to discover that the specification has been satisfied in a system in which infinitely many grants are issued, Using LTL$^{\triangledown}$, the designer can model vacuity and coverage in a way that would indicate the "level of alarm".

For example, keeping in mind that implications are disjunctions, we can refine the specification $\mathsf{G}(req \rightarrow \mathsf{F}grant)$ to $\mathsf{G}((\triangledown_{\frac{1}{3}} \neg req) \vee \mathsf{F}grant)$, which would get satisfaction value $\frac{1}{3}$ in computations in which it is satisfied only thanks to the fact there are only finitely many requests. As another example, recall the specification $\mathsf{G}(req \rightarrow (grant \vee \triangledown_{\frac{3}{4}} \mathsf{X}grant))$ in which we are happier if the request is granted immediately. In the context of sanity checks, we do not want two grants to be given to a single request. This can be formulated by the specification $\mathsf{G}(req \rightarrow ((grant \wedge \neg \mathsf{X}grant) \vee \triangledown_{\frac{3}{4}} (\neg grant \wedge \mathsf{X}grant) \vee \triangledown_{\frac{1}{2}} (grant \wedge \mathsf{X}grant)))$. Here, a request that is followed by two grants would decrease the satisfaction level to $\frac{1}{2}$.

Safety. Consider a vending machine that serves drinks. A natural safety property is to require the machine to always have drinks, say coffee and tea. Or should we say "coffee or tea"? Indeed, the quality of a safety property is closely related to the necessity operator. The operator allows us to value the level of violating the safety requirements. For example, we may require that always having some drinks is a must, while having all drinks is a nice to have, using the formula $\mathsf{G}((coffee \vee tea) \wedge \blacktriangledown_{\frac{3}{4}} (coffee \wedge tea))$. This example can obviously be generalized to provide the necessity level of each subset of drinks. Another safety requirement for the machine is to always have coins for change. Since this is not a critical requirement, we can formalize it by $\blacktriangledown_{\frac{1}{8}} \mathsf{G}coins$.

A.2 Proof of Theorem 1

Consider an LTL$^{\triangledown}$ formula φ, and let φ' be the LTL$^{\triangledown}$ formula obtained from φ by expressing all the $\blacktriangledown$ and $\blacktriangledown$ operators with $\triangledown$ (which introduces additive constants). We have to show that we can push inward all the $\neg$, $\triangledown$, and $+\kappa$ operators. Pushing can be done in an arbitrary order, and is done by applying the following equivalences.

- For pushing $\neg$ inwards:
  - $\neg\neg\varphi \equiv \varphi$

- $\neg(\varphi \wedge \psi) \equiv (\neg\varphi) \vee (\neg\psi)$
- $\neg(\varphi \vee \psi) \equiv (\neg\varphi) \wedge (\neg\psi)$
- $\neg\mathsf{X}\varphi \equiv \mathsf{X}\neg\varphi$
- $\neg(\varphi\mathsf{U}\psi) \equiv (\neg\varphi)\mathsf{R}(\neg\psi)$
- $\neg(\varphi\mathsf{R}\psi) \equiv (\neg\varphi)\mathsf{U}(\neg\psi)$
- $\neg\nabla_\lambda\varphi \equiv (1-\lambda) + \nabla_\lambda\neg\varphi$
- $\neg(\varphi + \kappa) \equiv \neg\varphi - \kappa$

– For pushing ∇ through a Boolean or temporal operator $\star$ (except for $\neg$):
- $\nabla_\lambda(\star\varphi) \equiv \star\nabla_\lambda\varphi$
- $\nabla_\lambda(\varphi \star \psi) \equiv \nabla_\lambda\varphi \star \nabla_\lambda\psi$
- $\nabla_\lambda\nabla_\mu\varphi \equiv \nabla_{\lambda\cdot\mu}\varphi$
- $\nabla_\lambda(\varphi + \kappa) \equiv \nabla_\lambda\varphi + (\lambda \cdot \kappa)$

– For pushing $+\kappa$ through a Boolean or temporal operator $\star$ (except for $\neg$ and ∇):
- $\kappa + (\star\varphi) \equiv \star(\kappa + \varphi)$
- $\kappa + (\varphi \star \psi) \equiv (\kappa + \varphi) \star (\kappa + \psi)$
- $\kappa + (\varphi + \kappa') \equiv \varphi + (\kappa + \kappa')$

A.3 Proof of Lemma 1

We prove the claim by induction on the structure of the formula.

– If φ is either `True` or `False`, then its satisfaction value in every computation is in $\{0, 1\}$.
– If φ is a literal $\kappa + \nabla_\lambda l$, then its satisfaction value is in $\{\kappa, \kappa + \lambda\}$.
– If $\varphi = \psi_1 \star \psi_2$, for $\star \in \{\vee, \wedge, \mathsf{U}, \mathsf{R}\}$, then the set of combined literals in φ is the union of the sets of combined literals in ψ_1 and ψ_2. Hence, by the induction hypothesis, the satisfaction value of ψ_1 and of ψ_2 in every computation, and in particular in suffixes of π, is in the finite set $\{0, 1, \kappa_1 + \lambda_1, \ldots, \kappa_n + \lambda_n, \kappa_1, \ldots, \kappa_n\}$. From this follows that all the $\max$ and $\min$ operations in *op* return a value in this finite set. So, from the semantics of *op*, the satisfaction value of φ is in this set too.
– If $\varphi = \mathsf{X}\psi$, then the combined literals in φ and ψ are the same. By the induction hypothesis, the satisfaction value of ψ in every computation, and in particular in π^1, is in the required set. Hence, so is the satisfaction value of φ in π.

A.4 Proof of Lemma 2

We denote by $[\![\pi, \varphi']\!]_B$ the Boolean satisfaction value of the LTL formula φ' in the computation π. Consider a computation π, an LTL^∇ formula φ, and a constant value $0 < v \leq 1$. Let $\varphi' = \mathrm{at\_least}(\varphi, v)$, and for every subformula ψ of φ, let $\psi' = \mathrm{at\_least}(\psi, v)$. We have to prove that $[\![\pi, \varphi']\!]_B = \mathtt{True}$ iff $[\![\pi, \varphi]\!] \geq v$. We do it by induction on the structure of φ.

– If φ is a combined literal, the claim follows directly from the definitions.
– If $\varphi = \psi_1 \vee \psi_2$, then $[\![\pi, \varphi']\!]_B = \mathtt{True}$ iff for all computations π, we have that $[\![\pi, \psi_1']\!]_B = \mathtt{True}$ or $[\![\pi, \psi_2']\!]_B = \mathtt{True}$. By the induction hypothesis, the latter holds iff either $[\![\pi, \psi_1]\!] \geq v$ or $[\![\pi, \psi_2]\!] \geq v$, and this holds iff $[\![\pi, \psi]\!] = \max([\![\pi, \psi_1]\!], [\![\pi, \psi_2]\!]) \geq v$.
– If $\varphi = \psi_1 \wedge \psi_2$, the proof is analogous to the case of $\psi_1 \vee \psi_2$.
– If $\varphi = \mathsf{X}\psi_1$, the proof follows directly from the definitions.

– If $\varphi = \psi_1 \mathsf{U} \psi_2$, we proceed as follows. If $[\![\pi, \varphi']\!]_B = \texttt{True}$ then for all computations π, we have that there is a position t, such that $[\![\pi^t, \psi_2']\!]_B = \texttt{True}$ and for every position $0 \leq j < t$, $[\![\pi^j, \psi_1']\!]_B = \texttt{True}$. Therefore, by the induction assumption, $[\![\pi^t, \psi_2]\!] \geq v$ and for every position $0 \leq j < t$, $[\![\pi^j, \psi_1]\!] \geq v$. Hence, $[\![\pi, \psi_1 \mathsf{U} \psi_2]\!] = \max_{0 \leq i < |\pi|} [\min([\![\pi^i, \psi_2]\!], \min_{0 \leq j < i} [\![\pi^j, \psi_1]\!])] \geq \min([\![\pi^t, \psi_2]\!], \min_{0 \leq j < i} [\![\pi^j, \psi_1]\!]) \geq v$.
  As for the other direction, $[\![\pi, \psi_1 \mathsf{U} \psi_2]\!] \geq v$ implies that there is a position t, such that $[\![\pi^t, \psi_2]\!] \geq v$ and for every position $0 \leq j < t$, $[\![\pi^j, \psi_1]\!] \geq v$. Therefore, $[\![\pi^t, \psi_2']\!]_B = \texttt{True}$ and for every position $0 \leq j < t$, $[\![\pi^j, \psi_1']\!]_B = \texttt{True}$. Hence, $[\![\pi, \varphi']\!]_B = \texttt{True}$.
– If $\varphi = \psi_1 \mathsf{R} \psi_2$, we proceed as follows. If $[\![\pi, \varphi']\!]_B = \texttt{True}$ then for all computations π, one of the following holds.
  - There is a position t, such that $[\![\pi^t, \psi_1']\!]_B = \texttt{True}$ and for every position $0 \leq j \leq t$, $[\![\pi^j, \psi_2']\!]_B = \texttt{True}$. Therefore, by the induction assumption, $[\![\pi^t, \psi_1]\!] \geq v$ and for every position $0 \leq j \leq t$, $[\![\pi^j, \psi_2]\!] \geq v$.
    Recall that $[\![\pi, \psi_1 \mathsf{R} \psi_2]\!] = \min_{0 \leq i < |\pi|} \xi$, where $\xi = \max([\![\pi^i, \psi_2]\!], \max_{0 \leq j < i} [\![\pi^j, \psi_1]\!])$. Accordingly, we have that $[\![\pi, \psi_1 \mathsf{R} \psi_2]\!] = \min(\xi_1, \xi_2)$, where $\xi_1 = \min_{0 \leq i < t} \xi$ and $\xi_2 = \min_{0 \leq t \leq < |\pi|} \xi$. Now, $\xi_1 \geq v$, since the $[\![\pi^i, \psi_2]\!]$ component is always bigger or equal to v, and $\xi_2 \geq v$, since the $\max_{0 \leq j < i} [\![\pi^j, \psi_1]\!]$ component is always bigger or equal to v. Hence, $[\![\pi, \psi_1 \mathsf{R} \psi_2]\!] \geq v$.
  - For every $0 \leq j < |\pi|$ it holds that $[\![\pi^j, \psi_2']\!]_B = \texttt{True}$. From the induction hypothesis follows that $[\![\pi^j, \psi_2]\!] \geq v$ (for every j as above), and in particular for every $0 \leq i < |\pi|$ it holds that $\max([\![\pi^i, \psi_2]\!], \max_{0 \leq j < i} [\![\pi^j, \psi_1]\!]) \geq v$. Thus, $[\![\pi, \psi_1 \mathsf{R} \psi_2]\!] \geq v$.

  As for the other direction, if $[\![\pi, \varphi']\!]_B = \texttt{False}$ then there is a position t, such that $[\![\pi^t, \psi_2']\!]_B = \texttt{False}$ and for every position $0 \leq j \leq t$, $[\![\pi^j, \psi_1']\!]_B = \texttt{False}$. Therefore, by the induction assumption, $[\![\pi^t, \psi_2]\!] < v$ and for every position $0 \leq j \leq t$, $[\![\pi^j, \psi_1]\!] < v$. Hence, $[\![\pi, \psi_1 \mathsf{R} \psi_2]\!] = \min_{0 \leq i < |\pi|} [\max([\![\pi^i, \psi_2]\!], \max_{0 \leq j < i} [\![\pi^j, \psi_1]\!])] \leq \max([\![\pi^t, \psi_2]\!], \max_{0 \leq j < t} [\![\pi^j, \psi_1]\!]) < v$.

A.5 Runs of weighted alternating automata

In order to define runs of alternating automata, we first have to define trees and weighted labeled trees. A *tree* is a prefix closed set $T \subseteq \mathbb{N}^*$ (i.e., if $x \cdot d \in T$, where $x \in \mathbb{N}^*$ and $d \in \mathbb{N}$, then also $x \in T$). The elements of T are called *nodes*. For every $x \in T$, the nodes $x \cdot d$ where $d \in \mathbb{N}$ are the *successors* of x. A node is a *leaf* if it has no successors. We sometimes refer to the length $|x|$ of x as its *level* in the tree. A *path* π of a tree T is a prefix-closed set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$, either x is a leaf or there exists a unique $d \in \mathbb{N}$ such that $x \cdot d \in \pi$.

An *edge* in T is a pair $\langle x, x \cdot d \rangle \in T \times T$. The set of edges in T is denoted $Edge(T)$. We sometimes refer to a path π as a sequence of edges. Then, we say that an edge $(x, x \cdot d) \in \pi$ iff $x, x \cdot d \in \pi$. Given an alphabet Σ, a *weighted Σ-labeled tree* is a triple $\langle T, V, C \rangle$, where T is a tree, $V : T \to \Sigma$ maps each node of T to a letter in Σ, and $C : Edge(T) \to \mathbb{Q}$ maps each edge of T to a cost in $\mathbb{Q}$.

A run of an alternating automaton is a Q-labeled weighted tree. Given a word $w = \sigma_1 \cdot \sigma_2 \cdots \sigma_n$, a run of $\mathcal{A}$ on w is a Q-labeled weighted tree $\tau = \langle T_r, r, \rho \rangle$, such that the following hold:

- $\varepsilon \in T_r$ and $r(\varepsilon) = q_0$.
- Consider a node $x \in T_r$ with $r(x) = q$ and $\delta(q, \sigma_{|x|+1}) = \theta$. There is a (possibly empty) set $S = \{(q_1, c_1), \ldots, (q_k, c_k)\} \subseteq 2^{Q \times \mathbb{Q}}$ such that S satisfies θ and for all $1 \leq d \leq k$, we have that $x \cdot d \in T_r$, $r(x \cdot d) = q_d$, and $\rho(x, x \cdot d) = c_d$.

The value of a run r, denoted $val(r)$, is the value of the minimal path in the run tree, and the value of a path in the run is the product of the weights along the path.

A run is accepting if all the paths in the run tree are accepting. That is, if all the paths end in an accepting state, in case of finite words, and if all paths visits α infinitely often (Büchi acceptance condition) in case of infinite words. Observe that a-priori, we may have runs where the infinite product of the weights does not converge. For the automata we work with here, this never happens.

A weighted automaton $\mathcal{A}$ assigns weights to (some of the) words in Σ^∞. The weight of a word w, denoted $\mathcal{A}(w)$, is the value of the maximal-valued accepting run of $\mathcal{A}$ on w. Formally, $\mathcal{A}(w) = \max\{val(r) : r \text{ is an accepting run of } \mathcal{A} \text{ on } w\}$. If there are no accepting runs of $\mathcal{A}$ on w, then $\mathcal{A}(w)$ is undefined. Thus, $\mathcal{A}$ can be thought of as a partial function from Σ^∞ to $\mathbb{Q}$.

We note that our weighted automata differ from a more common definition, in which the semantics of the automaton is determined by some semiring. For example, in automata based on the tropical semiring $\langle \mathbb{N} \cup \{\infty\}, +, \min, 0, \infty \rangle$ the weights are accumulated according to the $+$ operation, and the minimal run is taken. In our case, the underlying weight structure is not a semiring.

A.6 Correctness of the construction of $\mathcal{A}_\varphi$

We start with the case of finite words. There, $\mathcal{A}_\varphi$ is a WAFW and we have to prove that for every computation $\pi \in (2^{AP})^*$, it holds that $[\![\pi, \varphi]\!] = \mathcal{A}_\varphi(\pi)$. We denote by $\mathcal{A}^\psi_\varphi$ the automaton $\mathcal{A}_\varphi$ where we set the initial state to be ψ, for $\psi \in Q$. We prove that $[\![\pi, \psi]\!] = \mathcal{A}^\psi_\varphi(\pi)$, and we do it by an induction on the structure of ψ.

The only non-trivial cases in the induction are formulas of the form $\psi_1 \mathsf{U} \psi_2$ and $\psi_1 \mathsf{R} \psi_2$. Assume that $\psi = \psi_1 \mathsf{U} \psi_2$ and let π be a computation. By the induction hypothesis, and by the construction of the transition function, we see that $\mathcal{A}^\psi_\varphi(\pi) = \max\left\{\mathcal{A}^{\psi_2}_\varphi(\pi^1), \min\left\{\mathcal{A}^{\psi_1}_\varphi(\pi^1), \mathcal{A}^{\varphi}_\varphi(\pi^1)\right\}\right\}$. We prove by induction over $|\pi|$, that

$$\mathcal{A}^\psi_\varphi(\pi) = \max_{0 \leq i < |\pi|} [\min([\![\pi^i, \psi_2]\!], \min_{0 \leq j < i} [\![\pi^j, \psi_1]\!])].$$

The base case is $\mathcal{A}^\psi_\varphi(\pi) = [\![\pi^0, \psi_2]\!]$, which follows from the fact that the state $\psi_1 \mathsf{U} \psi_2$ is not in α, so any accepting run must go through ψ_2. We assume correctness for $|\pi| - 1$ and prove for $|\pi|$. By our observation above and by applying the induction hypothesis on π^1, it holds that

$$\mathcal{A}^\psi_\varphi(\pi) = \max\left\{\mathcal{A}^{\psi_2}_\varphi(\pi^1), \min\left\{\mathcal{A}^{\psi_1}_\varphi(\pi^1), \max_{1 \leq i < |\pi|} [\min([\![\pi^i, \psi_2]\!], \min_{1 \leq j < i} [\![\pi^j, \psi_1]\!])]\right\}\right\}.$$

We now observe the following property. For every collection of sets $\{B_i\}_{i=1}^k$ of numbers, and for every number v, it holds that

$$\min\left\{v, \max_i \min_{x \in B_i}(x)\right\} = \max_i \left\{\min_{x \in B_i}(v, x)\right\}.$$

It is not hard to verify the correctness of this property. Applying this to the equation above yields

$$\mathcal{A}^{\psi}_{\varphi}(\pi) = \max\left\{\mathcal{A}^{\psi_2}_{\varphi}(\pi^1), \max_{1\leq i<|\pi|} \min\left\{\mathcal{A}^{\psi_1}_{\varphi}(\pi^1), [\min([\![\pi^i, \psi_2]\!], \min_{1\leq j<i}[\![\pi^j, \psi_1]\!])]\right\}\right\}.$$

By aggregating the $\min$ and $\max$ we conclude the result. Dually, we can conclude the correctness of $\psi_1 \mathsf{R} \psi_2$.

We proceed to prove the correctness for infinite computations. The proof is similar to the case of finite words. However, we observe that the proof there used induction over $|\pi|$ to prove the correctness of the U and R operators. This cannot work in the case of infinite words, of course. To overcome this problems, we observe that for every formula $\psi = \psi_1 \mathsf{U} \psi_2$, there is an index $m \in \mathbb{N}$ such that

$$[\![\pi, \varphi]\!] = \mathcal{A}^{\psi}_{\varphi}(\pi) = \max_{0\leq i<|\pi|}[\min([\![\pi^i, \psi_2]\!], \min_{0\leq j<i}[\![\pi^j, \psi_1]\!])] = \max_{0\leq i<m}[\min([\![\pi^i, \psi_2]\!], \min_{0\leq j<i}[\![\pi^j, \psi_1]\!])].$$

That is, the value $[\![\pi, \varphi]\!]$ is determined after a finite prefix. This is a direct corollary of Lemma 1. Thus, we can use induction over m instead of induction over $|\pi|$ in order to proceed as in the proof of the finite case.

A.7 Proof of Theorem 4

We start with the case of finite words. Consider a very weak WAFW $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha\rangle$. We construct the simple WNFW $\mathcal{B} = \langle \Sigma, Q', \delta', Q'_0, \alpha'\rangle$ as follows. The set of states is $Q' = 2^Q \times \{0, 1, \lambda_1, ..., \lambda_k\}$, where $\{\lambda_1, ..., \lambda_k\}$ are the weights that appear on the edges of $\mathcal{A}$. Intuitively, the states of $\mathcal{B}$ are pairs $\langle S, m\rangle$ where S tracks the subset construction of $\mathcal{A}$ (as we do alternation removal in non-weighted automata) and m keeps track of the minimal weight encountered so far in the run. The set of initial states is $Q'_0 = \{\langle\{q_0\}, 1\rangle\}$. The set of accepting states is $\alpha' = 2^\alpha \times \{0, 1, \lambda_1, ..., \lambda_k\}$.

For every $\langle S, m\rangle \in Q'$ and $\sigma \in \Sigma$, we define δ' so that $(\langle S', m'\rangle, c) \in \delta'(\langle S, m\rangle, \sigma)$ iff S' is a set that satisfies $\delta(s, \sigma)$ for every $s \in S$, and m' is the minimum between m and the minimal value that appears in the transitions from $\delta(s, \sigma)$ to S' for some $s \in S$ (thus m' is the minimal value encountered so far in the run). Since we want the product of the weights read so far to be m', we define c to be 1 in case $m' = m$ (that is, no update to the m component takes place) and to be $\frac{m'}{m}$ in case $m' < m$ (that is, when the m component is updated, the multiplication of $\frac{m'}{m}$ by the weight m that has been calculated so far is m'). The above definition of δ' indeed guarantees that $\mathcal{B}$ keeps track of the minimal edge encountered in a path in the run-tree (nondeterministically guessing a run-tree).

The correctness of the construction follows from the following observations. First, the first components of the runs of $\mathcal{B}$ on a word are exactly the set of reachable sets in the different runs of $\mathcal{A}$ on the same word. This is easy to see, as in the subset construction for regular automata. Second, the second components of every run represent the minimal-weighted edge encountered so far in the run tree of $\mathcal{A}$ that corresponds to the run of $\mathcal{B}$. This is obvious from the way we define δ'.

We now proceed to the case of infinite words. The difference between this case and the case of finite words is analogous to the difference between alternation removal in regular automata and in very weak Büchi automata. In the case of finite words, we apply the subset construction, and equip it with an additional component to keep track of the minimal

weight encountered so far in the run. For infinite words, we can use the alternation removal construction from [4] (which is a simplification of the *breakpoint construction* [27] that utilizes the very weak nature of the automaton) and similarly equip it with a minimal-weight tracking component.

Finally, we observe that the alternation removal leaves us with a simple automaton. Indeed, visiting the same state twice along a run implies that in particular, the minimal weight encountered between the consecutive visits did not change. From the construction, we can see that a transition can have weight other than 1 only when it is from a state $\langle S, m\rangle$ to a state $\langle S', m\rangle$ such that $m' < m$. Furthermore, there is never a transition that increases the second component. Thus, the only way for a state to be visited twice is if every transition along the cycle has weight 1.

A.8 The product of a Kripke structure with a weighted automaton

Consider a Kripke structure $\mathcal{K} = \langle AP, S, I, \rho, L\rangle$ and a simple WNFW $\mathcal{A} = \langle Q^{\mathcal{A}}, 2^{AP}, \delta^{\mathcal{A}}, q_0^{\mathcal{A}}, \alpha^{\mathcal{A}}\rangle$. We define the product of $\mathcal{K}$ and $\mathcal{A}$ to be the WNFW $\mathcal{P} = \mathcal{K} \otimes \mathcal{A} = \langle Q, 2^{AP}, \delta, q_0, \alpha\rangle$ as follows. The states are $Q = S \times Q^{\mathcal{A}}$. We define a set of initial states $Q_0 = I \times \{q_0^{\mathcal{A}}\}$. We can modify the construction to have a single state in the standard way (add an initial state connected with an ϵ transition to all the initial state, and then preform ϵ-transition removal). The accepting states are $\alpha = S \times \alpha^{\mathcal{A}}$, and the transition function is defined as follows. For a state $\langle q, s\rangle$ and a letter $\sigma \in 2^{AP}$, we have that $(\langle q', s'\rangle, c) \in \delta(\langle q, s\rangle, \sigma)$ iff $(s, s') \in \rho$, $\sigma = L(s)$, and $(q', c) \in \delta^{\mathcal{A}}(q, \sigma)$.

The correctness of the standard product construction ensures us that for every word $w \in (2^{AP})^\infty$ it holds that $\mathcal{P}(w)$ is defined iff w is a computation of $\mathcal{K}$ and $\mathcal{A}(w)$ is defined. Furthermore, we notice that if w is a computation of $\mathcal{K}$, then the projection on the second component of every run of $\mathcal{P}$ on w is a run of $\mathcal{A}(w)$, and that every run of $\mathcal{A}$ on w is the projection of some run of $\mathcal{P}$ on w. Thus, the weight assigned to w by $\mathcal{P}$ is exactly $\mathcal{A}(w)$.

A.9 Deciding the existence of a heavy witness

We prove that given an automaton $\mathcal{P} = \mathcal{K} \times \mathcal{A}$ and a threshold t, where $\mathcal{A}$ is an automaton that was constructed from an $\mathrm{LTL}^{\triangledown}$ formula φ as per Corollary 1, the problem of deciding whether there is a word w such that $\mathcal{P}(w) > t$ can be solved in PSPACE in the size of φ and t, and NLOGSPACE in the size of $\mathcal{K}$. According to the semantics of our nondeterministic automata, it is sufficient to show that we can decide whether there exists a run of $\mathcal{P}$ whose value is more than t. In the case of finite words, it suffices to guess a simple path in $\mathcal{P}$ from the initial state to an accepting state, and make sure the multiplication of the edges along the run is greater than t. Indeed, since the weights are in $[0, 1]$, then a cycle in a path cannot increase the weight, so it is sufficient to consider simple paths.

Guessing a simple path can be done in NLOGSPACE in the size of $\mathcal{P}$, which is NPSPACE in the size of φ and NLOGSPACE in the size of $\mathcal{K}$. We have seen in the construction of $\mathcal{A}$ that the product of the weights along every run simply keeps track of the minimal weight encountered so far in the original (alternating) automaton. Thus, the set of weights that may appear as a product along a run is the set of weights that appear in the alternating automaton constructed from φ, which has size $O(|\varphi|)$. Thus, we only need to keep track of $O(|\varphi|)$ space for the weights. Since PSPACE=NPSPACE, we conclude the result.

In the case of infinite words, we need to find a simple path from the initial state to a state $q \in \alpha$, and a cycle from q to itself, such that the weight of the run obtained by reaching q

and repeating the cycle is greater than t. Recall that the automaton is simple, so the weight of every cycle is 1. Thus, it is sufficient to make sure that the multiplication of weights along the simple path to q is greater than t.

A.10 Alternating vs. nondeterministic weighted automata

Consider the WAFW $\mathcal{B}$ which we now describe. The state space is $\{s_0, s_p, s_q, \top\}$, the set of accepting states $\alpha = \{\top\}$, and the transition function $\delta(s, \sigma)$ is described in the table below.

$s \backslash \sigma$	$\emptyset$	$\{p\}$	$\{q\}$	$\{p, q\}$
s_0	$\langle s_p, \frac{1}{2}\rangle \wedge \langle s_q, \frac{1}{4}\rangle$	$\langle s_q, \frac{1}{4}\rangle \wedge \langle \top, 1\rangle$	$\langle s_p, \frac{1}{2}\rangle \wedge \langle \top, 1\rangle$	$\langle \top, 1\rangle$
s_p	$\langle s_p, \frac{1}{2}\rangle$	$\langle \top, 1\rangle$	$\langle s_p, \frac{1}{2}\rangle$	$\langle \top, 1\rangle$
s_q	$\langle s_q, \frac{1}{4}\rangle$	$\langle s_q, \frac{1}{4}\rangle$	$\langle \top, 1\rangle$	$\langle \top, 1\rangle$
$\top$	$\langle \top, 1\rangle$			

Intuitively, this automaton corresponds to the formula $\varphi = (\mathsf{F}_{\blacktriangledown \frac{1}{2}} p) \wedge (\mathsf{F}_{\blacktriangledown \frac{1}{4}} q)$, which is an $\blacktriangledown$-LTL formula with *discounting* operators. The weighted-eventuality operator $\mathsf{F}_{\blacktriangledown \frac{1}{2}} p$ assigns the value 2^{-i} to a word where p appears first in the i-th position.

Formally, it is easy to verify that for every computation π, it holds that

$$\mathcal{B}(\pi) = \min\left\{2^{-i}, 4^{-j} : p \in \pi_i, q \in \pi_j, 0 \leq i, j < |\pi|\right\}$$

We now show that there is no WNFW $\mathcal{A}$ such that for every $\pi \in (2^{\{p,q\}})^*$, it holds that $\mathcal{A}(\pi) = \mathcal{B}(\pi)$.

Assume by way of contradiction that there exists such a WNFW $\mathcal{A}$, and consider words of the form $\emptyset^k q \emptyset^l p$. Intuitively, $\mathcal{A}$ needs to check whether $l > k$ in order to determine whether the transition in each step should contribute to the product $\frac{1}{2}$ or $\frac{1}{4}$. This check cannot be done with automata. The formal proof is by a shrinking argument, with the exception of handling cycles whose (multiplicative) weight is in $(-1, 0)$. These pose a problem, which we handle by shrinking two occurrences of a cycle.

Formally, let c be the number of simple cycles in $\mathcal{A}$, and let n be the number of states in $\mathcal{A}$. Define $k = l = (c+1)(n+1)$, and consider the word $\pi = \emptyset^k\{q\}\emptyset^l\{p\}$. It holds that

$$\mathcal{A}(\pi) = \mathcal{B}(\pi) = \min\left\{\frac{1}{4}^{k+1}, \frac{1}{2}^{k+l+2}\right\} = \frac{1}{2}^{k+l+2}.$$

Also, observe that for every $j < k + l + 2$, it holds that

$$\frac{1}{4}^{k+1} < \frac{1}{2}^{j}.$$

Let r be an accepting run of $\mathcal{A}$ on π that attains the value $\frac{1}{2}^{k+l+2}$. By the semantics of WNFW, this implies that all other accepting runs have value of at least $\frac{1}{2}^{k+l+2}$. Let $r = r_0, ..., r_k, r_{k+1}, s_0, ..., s_l, s_{l+1}$. Since $l = (c+1)(n+1)$, then by the pigeonhole principle there exists a simple cycle that repeats twice in the $s_0, ..., s_l$ block. That is, there exist $i_0 < i_1 < j_0 < j_1$ such that $r_{i_0} = r_{i_1} = r_{j_0} = r_{j_1}$, and $(r_{i_0}, ..., r_{i_1}) = (r_{j_0}, ..., r_{j_1})$,

where the equality implies that the states are equal, and also the transition taken from state to state are the same, and have the same weight.

Denote by v the value of the cycle $(r_{i_0}, ..., r_{i_1})$. That is, the multiplication of the weights along the edges of the cycle. We distinguish between the following cases.

- If $v = 1$ or $v = -1$, then by pumping the cycle $(r_{i_0}, ..., r_{i_1})$ an even number of times, we can obtain a run with value $\frac{1}{2}^{k+l+2}$ for words of the form $\emptyset^k\{q\}\emptyset^j\{p\}$ for arbitrarily large j. Thus, the value assigned by $\mathcal{A}$ to these words is at least $\frac{1}{2}^{k+l+2}$, which is a contradiction.
- If $v > 1$ or $v < 1$, we can pump the cycle as above, and obtain a run with weight greater than 1, which is clearly a contradiction, since $\mathcal{B}$ never assigns a weight of more than 1.
- If $v = 0$ then the value of r is 0, which is an immediate contradiction.
- If $v \in (0, 1)$, we shrink the cycle $(r_{i_0}, ..., r_{i_1})$. The effect of this is that the new run r' has weight greater than $\frac{1}{2}^{k+l+2}$, and r' is an accepting run of $\mathcal{A}$ on the word $\emptyset^k\{q\}\emptyset^j\{p\}$, where $j < l$. As we stated above, the value of this word should be $\frac{1}{4}^{k+1}$, but the automaton assigns a greater weight, which is a contradiction.
- If $v \in (-1, 0)$, we would like to shrink as above. Unfortunately, this will negate the sign of the run, which means the value of the new run will be smaller. Here we use the second cycle $(r_{j_0}, ..., r_{j_1})$, which is the same as the first cycle. We shrink both cycles, thus keeping the sign, and obtaining the same contradiction as above.