Noname manuscript No.
(will be inserted by the editor)

The ins and outs of first-order runtime verification

Andreas Bauer[1,2] · **Jan-Christoph Küster**[1,3] ·
Gil Vegliach[1]

the date of receipt and acceptance should be inserted later

Abstract The main purpose of this paper is to introduce a first-order temporal logic, LTL^{FO}, and a corresponding monitor construction based on a new type of automaton, called spawning automaton.

Specifically, we show that monitoring a specification in LTL^{FO} boils down to an undecidable decision problem. The proof of this result revolves around specific ideas on what we consider a "proper" monitor. As these ideas are general, we outline them first in the setting of standard LTL, before lifting them to the setting of first-order logic and LTL^{FO}. Although due to the above result one cannot hope to obtain a complete monitor for LTL^{FO}, we prove the soundness of our automata-based construction and give experimental results from an implementation. These seem to substantiate our hypothesis that the automata-based construction leads to efficient runtime monitors whose size does not grow with increasing trace lengths (as is often observed in similar approaches). However, we also discuss formulae for which growth is unavoidable, irrespective of the chosen monitoring approach. Specifically, we provide a general categorisation of so called monitorable languages, which is closely related to this notion of "growth-inducing" (that is, trace-length dependent) formulae. It relates to the well-known safety-progress hierarchy, yet is orthogonal to it.

Keywords monitoring · spawning automata · temporal logic · first-order logic · monitorability · trace-length independence

1 Introduction

In the area of runtime verification (cf. [23,22,16,9]), a *monitor* typically describes a device or program which is automatically generated from a formal specification capturing undesired (resp. desired) system behaviour. The monitor's task is to passively observe a running system in order to detect if the behavioural specification has been satisfied or violated by the observed system behaviour. While,

[1]NICTA, Software Systems Research Group · [2] TU München, Software & Systems Engineering Group · [3]Australian National University, Logic & Computation Group

arguably, the majority of runtime verification approaches are based on propositional logic (or expressively conservative parametric extensions thereof; cf. §8 for an overview), there exist works that have considered full first-order logic (cf. [22, 6,4]). Monitoring first-order specifications has also gained prior attention in the database community, especially in the context of so called temporal triggers, which correspond to first-order temporal logic specifications that are evaluated wrt. a linear sequence of database updates (cf. [11,12,34]). Although the underlying logics are generally undecidable, the monitors in these works usually address decidable problems, such as "is the so far observed behaviour a violation of a given specification φ?" Additionally, in many approaches, φ must only ever be a safety or domain independent property for this problem to actually be decidable (cf. [11,4]), which can be ensured by syntactic restrictions on the input formula, for example.

As there exist many different ways in which a system can be monitored in this abstract sense, we are going to put forth very specific assumptions concerning the properties and inner-workings of what we consider a "proper" monitor. None of these assumptions is particularly novel or complicated, but they help describe and distinguish the task of a "proper" monitor from that of, say, a model checker, which can also be used to solve monitoring problems as we shall see.

The two basic assumptions are easy to explain: Firstly, we demand that a monitor is what we call *trace-length independent*, meaning that its efficiency does not decline with an increasing number of observations. Secondly, we demand that a monitor is *monotonic* wrt. reporting violations (resp. satisfication) of a specification, meaning that once the monitor returns "SAT" to the user, additional observations do not lead to it returning "UNSAT" (and vice versa). We are going to postulate further assumptions, but these are mere consequences of the two basic ones and are explained in §2.

At the heart of this paper, however, is a custom first-order temporal logic, in the following referred to as $\mathrm{LTL}^{\mathrm{FO}}$, which is undecidable. Yet we outline a sound, albeit incomplete, monitor construction for it based on a new type of automaton, called spawning automaton. $\mathrm{LTL}^{\mathrm{FO}}$ was originally developed for the specification of runtime verification properties of Android "Apps" and has already been used in that context (see [7] for details). Although [7] gave a monitoring algorithm for $\mathrm{LTL}^{\mathrm{FO}}$ based on formula rewriting, it turns out that the automata-based construction given in this paper leads to practically more efficient results in practice.

Structure of this paper. As our definition of what constitutes a "proper" monitor is not tied to a particular logic we will develop it first for standard LTL (§2), the quasi-standard in the area of runtime verification. In §3, we give a more detailed account of $\mathrm{LTL}^{\mathrm{FO}}$, before we lift the results of §2 to the first-order setting (§4). The automata-based monitor construction for $\mathrm{LTL}^{\mathrm{FO}}$ is given in §5 along with two comprehensive examples to convey an intuitive understanding of the underlying principles and ideas. In §6.1, we then discuss some optional optimisations to said algorithm, which aim at making the resulting monitor smaller, thus more efficient at runtime. They are discussed separately, because didactically it makes sense to separate underlying principles from implementation-oriented extensions. The section finishes, with experimental results, obtained from a Scala-implementation of our monitoring algorithm.

Arguably, §7 is a logical consequence of the previous sections, in that it discusses what languages can be effectively monitored (by our algorithms or any other for that matter) and how they relate to one another. As such, it lays out a formal language classification not unlike the well-known safety-progress hierarchy [28], but with a focus on monitoring.

Related work is discussed in §8, and §9 concludes. We have added some detailed proofs of formal statements in the appendix of this paper.

Previous versions of this paper. Since this is an extended version of a recent conference paper (see [8]), a word to highlight additional or new content is in order. Besides minor structural changes, the following contributions are unique in comparison with [8]:

- §6.1 on implementation optimisations.
- Extended experiments in §6.2, comparing the original and optimised versions of the monitor.
- An extensive example in §5, which explains the behaviour of the monitoring algorithm (in accordance to the earlier example in this section, which merely described the behaviour of spawning automata).
- §7 on monitorable and trace-length dependent formal languages.
- Detailed formal proofs.

2 Complexity of monitoring in the propositional case

In what follows, we assume basic familiarity with LTL and topics like model checking (cf. [13,3] for an overview). Despite that, let us first state a formal LTL semantics, since we will consider its interpretation on infinite and finite traces. For that purpose, let AP denote a set of propositions, LTL(AP) the set of well-formed LTL formulae over that set, and for some set X set $X^\infty = X^\omega \cup X^*$ to be the union of the sets of all infinite and finite traces over X. When AP is clear from the context or does not matter, we use LTL instead of LTL(AP). Also, for a given trace $w = w_0 w_1 \ldots$, the trace w^i is defined as $w_i w_{i+1} \ldots$. As a convention we use $u, u', \ldots$ to denote finite traces, by σ the trace of length 1, and w for infinite ones or where the distinction is of no relevance.

Definition 1 Let $\varphi \in \mathrm{LTL}(\mathrm{AP})$, $w \in (2^{\mathrm{AP}})^\infty$ be a non-empty trace, and $i \in \mathbb{N}_0$, then

$$\begin{aligned}
w^i \models p \ &\text{ iff } \ p \in w_i, \ \text{where } p \in \mathrm{AP},\\
w^i \models \neg\varphi \ &\text{ iff } \ w^i \models \varphi \text{ does not hold},\\
w^i \models \varphi \wedge \psi \ &\text{ iff } \ w^i \models \varphi \text{ and } w^i \models \psi,\\
w^i \models \mathsf{X}\varphi \ &\text{ iff } \ |w| > i \text{ and } w^{i+1} \models \varphi,\\
w^i \models \varphi \mathsf{U} \psi \ &\text{ iff } \ \text{there is a } k \text{ s.t. } i \leq k < |w|, w^k \models \psi, \ \text{and for all } i \leq j < k, w^j \models \varphi.
\end{aligned}$$

And if $w^0 \models \varphi$ holds, we usually write $w \models \varphi$ instead. Although this semantics, which was also proposed in [29], gives rise to mixed languages, i.e., languages consisting of finite and infinite traces, we shall only ever be concerning ourselves with either finite-trace or infinite-trace languages, but not mixed ones. It is easy to see that over infinite traces this semantics matches the definition of standard LTL. Recall, LTL is a decidable logic; in fact, the satisfiability problem for LTL is known to be PSpace-complete [33].

As there are no commonly accepted rules for what qualifies as a monitor (not even in the runtime verification community), there exist a myriad of different approaches to checking that an observed behaviour satisfies (resp. violates) a formal specification, such as an LTL formula. Some of these (cf. [23,6]) consist in solving the word problem (see Definition 2). A monitor following this idea can either first record the entire system behaviour in form of a trace $u \in \Sigma^+$, where Σ is a finite alphabet of events and Σ^+ the set of non-empty traces over said alphabet, or process the events incrementally as they are emitted by the system under scrutiny. Both approaches are documented in the literature (cf. [23,20,22,6]), but only the second one is suitable to detect property violations (resp. satisfaction) right when they occur.

Definition 2 The *word problem for* LTL is defined as follows.
Input: A formula $\varphi \in \mathrm{LTL}(\mathrm{AP})$ and some trace $u \in (2^{\mathrm{AP}})^+$.
Question: Does $u \models \varphi$ hold?

In [29], a bilinear algorithm for this problem was presented (an even more efficient solution was recently given in [26]). Hence, the first sort of monitor, which is really more of a test oracle than a monitor, solves a classical decision problem. The second sort of monitor, however, solves an entirely different kind of problem, which cannot be stated in complexity-theoretical terms at all: its input is an LTL formula and a finite albeit unbounded trace which *grows* incrementally. This means that this monitor solves the word problem for each and every new event that is added to the trace at runtime. We can therefore say that the word problem acts as a lower bound on the complexity of the monitoring problem that such a monitor solves; or, in other words, the problem that the online monitor solves is at least as hard as the problem that the offline monitor solves.

There are approaches to build efficient (i.e., trace-length independent) monitors that repeatedly answer the word problem (cf. [23]). However, such approaches violate our second basic assumption, mentioned in the introduction, in that they are necessarily non-monotonic. To see this, consider $\varphi = a \mathsf{U} b$ and some trace $u = \{a\}\{a\}\dots\{a\}$ of length n. Using our finite-trace interpretation, $u \not\models \varphi$. However, if we add $u_{n+1} = \{b\}$, we get $u \models \varphi$.[1] For the user, this essentially means that she cannot trust the verdict of the monitor as it may flip in the future, unless of course it is obvious from the start that, e.g., only safety properties are monitored and the monitor is built merely to detect violations, i.e., bad prefixes. However, if we take other monitorable languages into account as we do in this paper, i.e., those that have either good or bad prefixes (or both), we need to distinguish between satisfaction and violation of a property (and want the monitor to report either occurrence truthfully).

Definition 3 For any $L \subseteq \Sigma^\omega$, $u \in \Sigma^*$ is called a *good prefix* (resp. *bad prefix*) of L iff $u\Sigma^\omega \subseteq L$ holds (resp. $u\Sigma^\omega \cap L = \emptyset$), where $u\Sigma^\omega$ is the set of infinite traces beginning with u.

We shall use $\mathrm{good}(L) \subseteq \Sigma^*$ (resp. $\mathrm{bad}(L)$) to denote the set of good (resp. bad) prefixes of L. For brevity, we also write $\mathrm{good}(\varphi)$ instead of $\mathrm{good}(\mathcal{L}(\varphi))$, and do the same for $\mathrm{bad}(\mathcal{L}(\varphi))$.

[1] Note that this effect is not particular to our choice of finite-trace interpretation. Had we used, e.g., what is known as the weak finite-trace semantics, discussed in [18], we would first have had $u \models \varphi$ and if $u_{n+1} = \emptyset$, subsequently $u \not\models \varphi$.

A monitor that detects good (resp. bad) prefixes has been termed impartial in [16] as it not only states something about the past, but also about the future: once a good (resp. bad) prefix has been detected, no matter how the system would evolve in an indefinite future, the property would remain satisfied (resp. violated). In that sense, impartial monitors are monotonic by definition. A further monitor characteristic is anticipation [16], which demands detection of shortest good or bad prefixes. While in [9] a construction is given that yields a trace-length independent (even optimal) impartial and anticipatory monitor for an LTL formula as well as a timed extension called TLTL, we shall see that obtaining anticipatory monitors for first-order temporal specifications is generally impossible. The obtained monitor basically returns $\top$ to the user if $u \in \mathrm{good}(\varphi)$ holds, $\perp$ if $u \in \mathrm{bad}(\varphi)$ holds, and ? otherwise. Not surprisingly though, the monitoring problem such a monitor solves is computationally more involved than the word problem. It solves what we call the prefix problem (of LTL), which can easily be shown PSpace-complete by way of LTL satisfiability.

Definition 4 The *prefix problem for* LTL is defined as follows.
Input: A formula $\varphi \in \mathrm{LTL(AP)}$ and some trace $u \in (2^{\mathrm{AP}})^*$.
Question: Does $u \in \mathrm{good}(\varphi)$ (resp. $\mathrm{bad}(\varphi)$) hold?

Theorem 1 *The prefix problem for* LTL *is PSpace-complete.*

Proof For brevity, we will only focus on bad prefixes. It is easy to see that $u \in \mathrm{bad}(\varphi)$ iff $\mathcal{L}(u_0 \wedge \mathsf{X}u_1 \wedge \mathsf{XX}u_2 \wedge \ldots \wedge \varphi) = \emptyset$. Constructing this conjunction takes polynomial time and the corresponding emptiness check can be performed in PSpace [33]. For hardness, we proceed with a reduction of LTL satisfiability. Again, it is easy to see that $\mathcal{L}(\varphi) \neq \emptyset$ iff $\sigma \notin \mathrm{bad}(\mathsf{X}\varphi)$ for any $\sigma \in 2^{\mathrm{AP}}$. This reduction is linear, and as PSpace = co-PSpace, the statement follows. $\square$

We would like to point out the possibility of building an impartial and anticipatory though trace-length dependent LTL monitor using an "off the shelf" model checker, which accepts a propositional Kripke structure and an LTL formula as input. Note that here we make the common assumption that Kripke structures produce infinite as opposed to finite traces.

Definition 5 The *model checking problem for* LTL is defined as follows.
Input: A formula $\varphi \in \mathrm{LTL(AP)}$ and a Kripke structure $\mathcal{K}$ over 2^{AP}.
Question: Does $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi)$ hold?

As in LTL the model checking and the satisfiability problems are both PSpace-complete [33], we can use a model checking tool as monitor: given that it is straightforward to construct $\mathcal{K}$, s.t. $\mathcal{L}(\mathcal{K}) = u(2^{\mathrm{AP}})^\omega$, in no more than polynomial time, we return $\top$ to the user if $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi)$ holds, $\perp$ if $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\neg\varphi)$ holds, and ? if neither holds. One could therefore be tempted to think of monitoring merely in terms of a model checking problem, but we shall see that as soon as the logic in question has an undecidable satisfiability problem this reduction fails. Besides, it can be questioned whether monitoring as model checking leads to a desirable monitor with its obvious trace-length dependence and having to repeatedly solve a PSpace-complete problem for each new event.

3 LTL^{FO}—Formal definitions and notation

Let us now introduce our first-order specification language LTL^{FO} and related concepts in more detail. The first concept we need is that of a *sorted first-order signature*, given as $\Gamma = (\mathbf{S}, \mathbf{F}, \mathbf{R})$, where $\mathbf{S}$ is a finite non-empty set of sorts, $\mathbf{F}$ a finite set of function symbols and $\mathbf{R} = \mathbf{U} \cup \mathbf{I}$ a finite set of a priori uninterpreted and interpreted predicate symbols, s.t. $\mathbf{U} \cap \mathbf{I} = \emptyset$ and $\mathbf{R} \cap \mathbf{F} = \emptyset$. The former set of predicate symbols are referred to as $\mathbf{U}$-operators and the latter as $\mathbf{I}$-operators. As is common, 0-ary functions symbols are also referred to as constant symbols. We assume that all operators in Γ have a given arity that ranges over the sorts given by $\mathbf{S}$, respectively. We also assume an infinite supply of variables, $\mathbf{V}$, that also range over $\mathbf{S}$ and where $\mathbf{V} \cap (\mathbf{F} \cup \mathbf{R}) = \emptyset$. Let us refer to the first-order language determined by Γ as $\mathcal{L}(\Gamma)$. While *terms* in $\mathcal{L}(\Gamma)$ are made up of variables and function symbols, *formulae* of $\mathcal{L}(\Gamma)$ are defined as follows:

$$\varphi ::= p(t_1, \dots, t_n) \mid r(t_1, \dots, t_n) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \varphi \mathsf{U} \varphi \mid \forall(x_1, \dots, x_n) : p.\ \varphi,$$

where $t_1, \dots, t_n$ are terms, $p \in \mathbf{U}$, $r \in \mathbf{I}$, and $x_1, \dots, x_n \in \mathbf{V}$. As variables are sorted, in the quantified formula $\forall(x_1, \dots, x_n) : p.\ \varphi$, the $\mathbf{U}$-operator p with arity $\tau_1 \times \dots \times \tau_n$, defines the sorts of variables $x_1, \dots, x_n$ to be $\tau_1, \dots, \tau_n$, with $\tau_i \in \mathbf{S}$, respectively. For terms $t_1, \dots, t_n$, we say that $p(t_1, \dots, t_n)$ is well-sorted if the sort of every t_i is τ_i. This notion is inductively applicable to terms. Moreover, we consider only well-sorted formulae and refer to the set of all well-sorted $\mathcal{L}(\Gamma)$ formulae over a signature Γ in terms of $\text{LTL}^{\text{FO}}_{\Gamma}$. When a specific Γ is either irrelevant or clear from the context, we will simply write LTL^{FO} instead. When convenient and a certain index is of no importance in the given context, we also shorten notation of a vector $(x_1, \dots, x_n)$ by a (bold) $\mathbf{x}$.

A *Γ-structure*, or just *first-order structure* is a pair $\mathfrak{A} = (|\mathfrak{A}|, I)$, where $|\mathfrak{A}| = |\mathfrak{A}|_1 \cup \dots \cup |\mathfrak{A}|_n$, is a non-empty set called domain, s.t. every sub-domain $|\mathfrak{A}|_i$ is either a non-empty finite or countable set (e.g., set of all integers or strings) and I an interpretation. I assigns to each sort $\tau_i \in \mathbf{S}$ a specific sub-domain $\tau_i^I = |\mathfrak{A}|_i$, to each function symbol $f \in \mathbf{F}$ of arity $\tau_1 \times \dots \times \tau_l \longrightarrow \tau_m$ a function $f^I : |\mathfrak{A}|_1 \times \dots \times |\mathfrak{A}|_l \longrightarrow |\mathfrak{A}|_m$, and to every $\mathbf{I}$-operator r with arity $\tau_1 \times \dots \times \tau_m$ a relation $r^I \subseteq |\mathfrak{A}|_1 \times \dots \times |\mathfrak{A}|_m$. We restrict ourselves to computable relations and functions. In that regard, we can think of I as a mapping between $\mathbf{I}$-operators (resp. function symbols) and the corresponding algorithms which compute the desired return values, each conforming to the symbols' respective arities. Note that the interpretation of $\mathbf{U}$-operators is rather different from $\mathbf{I}$-operators, as it is closely tied to what we call a trace and therefore discussed after we introduce the necessary notions and notation.

We model observed system behaviour in terms of *actions*: Let $p \in \mathbf{U}$ with arity $\tau_1 \times \dots \times \tau_m$ and $\mathbf{d} \in D_p = |\mathfrak{A}|_1 \times \dots \times |\mathfrak{A}|_m$, then we call $(p, \mathbf{d})$ an *action*. We refer to finite sets of actions as *events*. A system's behaviour is therefore a finite *trace* of events, which we also denote as a sequence of sets of ground terms $\{sms(1234)\}\{login(\text{"user"})\}\dots$ when we mean the sequence of tuples $\{(sms, 1234)\}$ $\{(login, \text{"user"})\}\dots$ Therefore the occurrence of some action $sms(1234)$ in the trace at position $i \in \mathbb{N}_0$, written $sms(1234) \in w_i$, indicates that, at time i, $sms(1234)$ holds (or, from a practical point of view, an SMS was sent to number 1234). We follow the convention that only symbols from $\mathbf{U}$ appear in a trace, which therefore gives these symbols their respective interpretations. The following formalises this notion.

A *first-order temporal structure* is a tuple $(\overline{\mathfrak{A}}, w)$, where $\overline{\mathfrak{A}} = (|\mathfrak{A}_0|, I_0)(|\mathfrak{A}_1|, I_1)$ $\dots$ is a (possibly infinite) sequence of first-order structures and $w = w_0 w_1 \dots$ a corresponding trace. We demand that for all $\mathfrak{A}_i$ and $\mathfrak{A}_{i+1}$ from $\overline{\mathfrak{A}}$, it is the case that $|\mathfrak{A}_i| = |\mathfrak{A}_{i+1}|$ for all $f \in \mathbf{F}$, $f^{I_{i+1}} = f^{I_i}$, and for all $\tau \in \mathbf{S}$, $\tau^{I_i} = \tau^{I_{i+1}}$. For any two structures, $\mathfrak{A}$ and $\mathfrak{A}'$, which satisfy these conditions, we write $\mathfrak{A} \sim \mathfrak{A}'$. Moreover given some $\overline{\mathfrak{A}}$ and $\mathfrak{A}$, if for all $\mathfrak{A}_i$ from $\overline{\mathfrak{A}}$, we have that $\mathfrak{A}_i \sim \mathfrak{A}$, we also write $\overline{\mathfrak{A}} \sim \mathfrak{A}$. In other words, the latter notation states that the same domain as well as interpretation of functions and sorts, defined in $\mathfrak{A}$, is used throughout all first-order structures of the sequence $\overline{\mathfrak{A}}$. Finally, the interpretation of a **U**-operator p with arity $\tau_1 \times \dots \times \tau_m$ is then defined wrt. a position i in w as $p^{I_i} = \{\mathbf{d} \mid (p, \mathbf{d}) \in w_i\}$. Essentially this means that, unlike function symbols, **U**- and **I**-operators don't have to be rigid.

Note also that from this point forward, we consider only the case where the policy to be monitored is given as a closed formula, i.e., a sentence. This is closely related to our means of quantification: a quantifier in LTL$^{\text{FO}}$ is restricted to those elements that appear in the trace, and not arbitrary elements from a (possibly infinite) domain. While certain policies cannot be expressed with this restriction (e.g., "for all phone numbers x that are not in the contact list, $r(x)$ is true"), this restriction bears the advantage that, when examining a given trace, functions and relations are only ever evaluated over known objects. The advantages of this type of quantification in monitoring first-order languages have also been pointed out in [22,6]. In other words, had we allowed free variables, a monitor might end up having to "try out" all the different domain elements in order to evaluate such policies, which runs counter to our design rationale of quantification.

In what follows, let us fix a particular Γ. The semantics of LTL$^{\text{FO}}$ can now be defined wrt. a quadruple $(\overline{\mathfrak{A}}, w, v, i)$ as follows, where $i \in \mathbb{N}_0$, and v is an (initially empty) set of valuations assigning domain values to variables:

$$
\begin{array}{rll}
(\overline{\mathfrak{A}}, w, v, i) \models p(t_1, \dots, t_n) & \text{iff} & (t_1^{I_i}, \dots, t_n^{I_i}) \in p^{I_i}, \\
(\overline{\mathfrak{A}}, w, v, i) \models r(t_1, \dots, t_n) & \text{iff} & (t_1^{I_i}, \dots, t_n^{I_i}) \in r^{I_i}, \\
(\overline{\mathfrak{A}}, w, v, i) \models \neg\varphi & \text{iff} & (\overline{\mathfrak{A}}, w, v, i) \models \varphi \text{ is not true}, \\
(\overline{\mathfrak{A}}, w, v, i) \models \varphi \wedge \psi & \text{iff} & (\overline{\mathfrak{A}}, w, v, i) \models \varphi \text{ and } (\overline{\mathfrak{A}}, w, v, i) \models \psi, \\
(\overline{\mathfrak{A}}, w, v, i) \models \mathsf{X}\varphi & \text{iff} & |w| > i \text{ and } (\overline{\mathfrak{A}}, w, v, i+1) \models \varphi, \\
(\overline{\mathfrak{A}}, w, v, i) \models \varphi \mathsf{U} \psi & \text{iff} & \text{for some } k \geq i, (\overline{\mathfrak{A}}, w, v, k) \models \psi, \\
 & & \text{and } (\overline{\mathfrak{A}}, w, v, j) \models \varphi \text{ for all } i \leq j < k, \\
(\overline{\mathfrak{A}}, w, v, i) \models \forall (x_1, \dots, x_n) : p.\ \varphi & \text{iff} & \text{for all } (p, d_1, \dots, d_n) \in w_i, \\
 & & (\overline{\mathfrak{A}}, w, v \cup \{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}, i) \models \varphi,
\end{array}
$$

where terms are evaluated inductively and x^I treated as $v(x)$. If $(\overline{\mathfrak{A}}, w, v, 0) \models \varphi$, we write $(\overline{\mathfrak{A}}, w, v) \models \varphi$, and if v is irrelevant or clear from the context, $(\overline{\mathfrak{A}}, w) \models \varphi$.

Later we will also make use of the (possibly infinite) set of all actions (resp. events) wrt. $\mathfrak{A}$, given as $(\mathfrak{A})$-Act $= \bigcup_{p \in \mathbf{U}} \{(p, \mathbf{d}) \mid \mathbf{d} \in D_p\}$ (resp. $(\mathfrak{A})$-Ev $= 2^{\text{Act}}$) and take the liberty to omit the trailing $(\mathfrak{A})$ whenever a particular $\mathfrak{A}$ is either irrelevant or clear from the context. We can then describe the *generated language* of φ, $\mathcal{L}(\varphi)$ (or simply the language of φ, i.e., the set of all logical models of φ) compactly as $\mathcal{L}(\varphi) = \{(\overline{\mathfrak{A}}, w) \mid w_i \in \text{Ev} \text{ and } (\overline{\mathfrak{A}}, w) \models \varphi\}$, although, as before, we shall only ever concern ourselves with either infinite- or finite-trace languages, but not mixed ones. Finally, we will use common syntactic "sugar", including $\exists(x_1,\ldots,x_n):p.\ \varphi = \neg(\forall(x_1,\ldots,x_n):p.\ \neg\varphi)$, etc.

Example 1 For brevity, cf. [7] for some example policies formalised in LTL$^{\text{FO}}$. However, to give at least an intuition, let's pick up the idea of monitoring Android "Apps" again, and specify that "Apps" must not send SMS messages to numbers not in a user's contact database. Assuming there exists a **U**-operator *sms*, which is true / appears in the trace, whenever an "App" sends an SMS message to phone number x, we could formalise said policy in terms of $\mathsf{G}\forall x : \mathit{sms}.\ \mathit{contact}(x)$. Note how in this formula the meaning of x is given implicitly by the arity of *sms* and must match the definition of *contact* in each world, i.e., x is not just any domain element, but a numerical value, which *sms* uses to capture the phone number associated with an incoming SMS. Also note how *sms* itself is interpreted indirectly via its occurrence in the trace, whereas *contact* never appears in the trace, even if true. *contact* can be thought of as interpreted via a program that queries a user's contact database, whose contents may change over time.

4 Complexity of monitoring in the first-order case

LTL$^{\text{FO}}$ as defined above is undecidable as can be shown by way of the following lemma. It basically helps us reduce finite satisfiability of standard first-order logic to LTL$^{\text{FO}}$.

Lemma 1 *Let φ be a sentence in first-order logic, then we can construct a corresponding $\psi \in$ LTL$^{\text{FO}}$ s.t. φ has a finite model iff ψ is satisfiable.*

Theorem 2 LTL$^{\text{FO}}$ *is undecidable.*

Proof (Sketch) Follows from Lemma 1 and Trakhtenbrot's Theorem (cf. [27, §9]).

Let's now define what is meant by Kripke structures in our new setting. They either give rise to infinite-trace languages (i.e., have a left-total transition relation), or represent finite traces (i.e, each state has at most one successor and the transition relation is loop-free). The latter ones we refer to as linear Kripke structures. For brevity, we shall restrict to the definition of the former. Note that we will also skip detailed redefinitions of the decision problems discussed in §2, since the concepts transfer in a straightforward manner.

Definition 6 Given some $\mathfrak{A}$, a *($\mathfrak{A}$)-Kripke structure*, or just first-order Kripke structure, is a state-transition system $\mathcal{K} = (S, s_0, \lambda, \rightarrow)$, where S is a finite set of states, $s_0 \in S$ a distinguished initial state, $\lambda : S \longrightarrow \widehat{\mathfrak{A}} \times \mathrm{Ev}$, where $\widehat{\mathfrak{A}} = \{\mathfrak{A}' \mid \mathfrak{A}' \sim \mathfrak{A}\}$, a labelling function, and $\rightarrow \subseteq S \times S$ a (left-total) transition relation.

Definition 7 For a ($\mathfrak{A}$)-Kripke structure $\mathcal{K}$ with states $s_0,\ldots,s_n$, the *generated language* is given as $\mathcal{L}(\mathcal{K}) = \{(\overline{\mathfrak{A}}, w) \mid (\mathfrak{A}_0, w_0) = \lambda(s_0)$ and for all $i \in \mathbb{N}$ there exist some $j,k \in \{0,\ldots,n\}$ s.t. $(\mathfrak{A}_i, w_i) = \lambda(s_j), (\mathfrak{A}_{i-1}, w_{i-1}) = \lambda(s_k)$ and $(s_k, s_j) \in \rightarrow\}$.

The inputs to the LTL$^{\text{FO}}$ word problem are therefore an LTL$^{\text{FO}}$ formula and a linear first-order Kripke structure, representing a finite input trace. Unlike in standard LTL,

Theorem 3 *The word problem for* LTL^{FO} *is PSpace-complete.*

The inputs to the LTL^{FO} model checking problem, in turn, are a left-total first-order Kripke structure, which gives rise to an infinite-trace language, and an LTL^{FO} formula.

Theorem 4 *The model checking problem for* LTL^{FO} *is in ExpSpace.*

The reason for this result is that we can devise a reduction of that problem to LTL model checking in exponential space. While the PSpace-lower bound is easy, e.g., via reduction of the LTL^{FO} word problem, we currently do not know how tight these bounds are and, therefore, leave this as an open problem. Note also that the results of both Theorem 3 and Theorem 4 are obtained even without taking into account the complexities of the interpretations of function symbols and **I**-operators; that is, for these results to hold, we assume that interpretations do not exceed polynomial, resp. exponential space.

We have seen in §2 that the prefix problem lies at the heart of an impartial monitor. While in LTL it was possible to build an impartial and anticipatory monitor using a model checker (albeit a very inefficient one), the following shows that this is no longer possible.

Lemma 2 *Let* $\mathfrak{A}$ *be a first-order structure and* $\varphi \in \text{LTL}^{\text{FO}}$*, then* $\mathcal{L}(\varphi)_{\mathfrak{A}} = \{(\overline{\mathfrak{A}}, w) \mid \overline{\mathfrak{A}} \sim \mathfrak{A}, w \in \text{Ev}^{\omega}, \text{ and } (\overline{\mathfrak{A}}, w) \models \varphi\}$*. Testing if* $\mathcal{L}(\varphi)_{\mathfrak{A}} \neq \emptyset$ *is generally undecidable.*

Theorem 5 *The prefix problem for* LTL^{FO} *is undecidable.*

Proof (Sketch) As in Theorem 1: $(\mathfrak{A}, \sigma) \in \text{bad}(\mathsf{X}\varphi)$ iff $\mathcal{L}(\varphi)_{\mathfrak{A}} = \emptyset$ for any $\sigma \in \text{Ev}$.

5 Monitoring LTL^{FO}

A corollary of Theorem 5 is that there cannot exist a complete monitor for LTL^{FO}-definable infinite trace languages. Yet one of the main contributions of our work is to show that one can build a sound and efficient LTL^{FO} monitor using a new kind of automaton. Before we go into the details of the actual monitoring algorithm, let us first consider the automaton model, which we refer to as *spawning automaton* (SA). SAs are called that, because when they process their input, they potentially "spawn" a positive Boolean combination of "children SAs" (i.e., subautomata) in each such step. Let $\mathcal{B}^+(X)$ denote the set of all positive Boolean formulae over the set X. We say that some set $Y \subseteq X$ satisfies a formula $\beta \in \mathcal{B}^+(X)$, written $Y \models \beta$, if the truth assignment that assigns true to all elements in Y and false to all $X - Y$ satisfies β.

Definition 8 A *spawning automaton*, or simply SA, is given by $\mathcal{A} = (\Sigma, l, Q, Q_0, \delta_{\rightarrow}, \delta_{\downarrow}, \mathcal{F})$, where Σ is a countable set called alphabet, $l \in \mathbb{N}_0$ the level of $\mathcal{A}$, Q a finite set of states, $Q_0 \subseteq Q$ a set of distinguished initial states, $\delta_{\rightarrow}$ a transition relation, $\delta_{\downarrow}$ what is called a spawning function, and $\mathcal{F} = \{F_1, \ldots, F_n \mid F_i \subseteq Q\}$ an acceptance condition (to be defined later on). We have $\delta_{\rightarrow} : Q \times \Sigma \longrightarrow 2^Q$ and $\delta_{\downarrow} : Q \times \Sigma \longrightarrow \mathcal{B}^+(\mathcal{A}^{<l})$, where $\mathcal{A}^{<l} = \{\mathcal{A}' \mid \mathcal{A}' \text{ is an SA with level less than } l\}$.

Definition 9 A *run* of spawning automaton $\mathcal{A}$ over input $w \in \Sigma^\omega$ is a mapping $\rho : \mathbb{N}_0 \longrightarrow Q$, s.t. $\rho(0) \in Q_0$ and $\rho(i+1) \in \delta_\rightarrow(\rho(i), w_i)$ for all $i \in \mathbb{N}_0$. ρ is *locally accepting* if $\mathrm{Inf}(\rho) \cap F_i \neq \emptyset$ for all $F_i \in \mathcal{F}$, where $\mathrm{Inf}(\rho)$ denotes the set of states visited infinitely often. It is called *accepting* if $l = 0$ and it is locally accepting. If $l > 0$, ρ is called accepting if it is locally accepting and for all $i \in \mathbb{N}_0$ there is a set $Y \subseteq \mathcal{A}^{<l}$, s.t. $Y \models \delta_\downarrow(\rho(i), w_i)$ and all spawning automata $\mathcal{A}' \in Y$ have an accepting run, ρ', over w^i. The *accepted language* of $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, consists of all $w \in \Sigma^\omega$, for which it has at least one accepting run.

5.1 Spawning automata construction

Given some $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$, let us now examine in detail how to build the corresponding SA, $\mathcal{A}_\varphi = (\Sigma, l, Q, Q_0, \delta_\rightarrow, \delta_\downarrow, \mathcal{F})$ s.t. $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ holds. To this end, we set $\Sigma = \{(\mathfrak{A}, \sigma) \mid \sigma \in (\mathfrak{A})\text{-}\mathrm{Ev}\}$. If φ is not a sentence, we write $\mathcal{A}_{\varphi,v}$ to denote the spawning automaton for φ in which free variables are mapped according to a finite set of valuations v.[2] To define the set of states for an SA, we make use of a restricted subformula function, $\mathrm{sf}|_\forall(\varphi)$, which is defined like a generic subformula function, except if φ is of the form $\forall \mathbf{x} : p.\ \psi$, we have $\mathrm{sf}|_\forall(\varphi) = \{\varphi\}$. This essentially means that an SA for a formula φ on the topmost level looks like the generalised Büchi automaton (GBA, cf. [3]) for φ, where quantified subformulae have been interpreted as atomic propositions.

For example, if $\varphi = \psi \wedge \forall \mathbf{x} : p.\ \psi'$, where ψ is a quantifier-free formula, then $\mathcal{A}_\varphi$, at the topmost level n, is like the GBA for the LTL formula $\psi \wedge a$, where a is an atomic proposition; or in other words, $\mathcal{A}_\varphi$ handles the subformula $\forall \mathbf{x} : p.\ \psi'$ separately in terms of a subautomaton of level $n-1$ (see also definition of $\delta_\downarrow$ below).

Finally, we define the closure of φ wrt. $\mathrm{sf}|_\forall(\varphi)$ as $\mathrm{cl}(\varphi) = \{\neg\psi \mid \psi \in \mathrm{sf}|_\forall(\varphi)\} \cup \mathrm{sf}|_\forall(\varphi)$, i.e., the smallest set containing $\mathrm{sf}|_\forall(\varphi)$, which is closed under negation. The *set of states* of $\mathcal{A}_\varphi$, Q, consists of all complete subsets of $\mathrm{cl}(\varphi)$; that is, a set $q \subseteq \mathrm{cl}(\varphi)$ is complete iff

- for any $\psi \in \mathrm{cl}(\varphi)$ either $\psi \in q$ or $\neg\psi \in q$, but not both; and
- for any $\psi \wedge \psi' \in \mathrm{cl}(\varphi)$, we have that $\psi \wedge \psi' \in q$ iff $\psi \in q$ and $\psi' \in q$; and
- for any $\psi \mathsf{U} \psi' \in \mathrm{cl}(\varphi)$, we have that if $\psi \mathsf{U} \psi' \in q$ then $\psi' \in q$ or $\psi \in q$, and if $\psi \mathsf{U} \psi' \notin q$, then $\psi' \notin q$.

Let $q \in Q$ and $\mathfrak{A} = (|\mathfrak{A}|, I)$. The *transition function* $\delta_\rightarrow(q, (\mathfrak{A}, \sigma))$ is defined iff

- for all $p(\mathbf{t}) \in q$, we have $\mathbf{t}^I \in p^I$ and for all $\neg p(\mathbf{t}) \in q$, we have $\mathbf{t}^I \notin p^I$,
- for all $r(\mathbf{t}) \in q$, we have $\mathbf{t}^I \in r^I$ and for all $\neg r(\mathbf{t}) \in q$, we have $\mathbf{t}^I \notin r^I$.

In which case, for any $q' \in Q$, we have that $q' \in \delta_\rightarrow(q, (\mathfrak{A}, \sigma))$ iff

- for all $\mathsf{X}\psi \in \mathrm{cl}(\varphi)$, we have $\mathsf{X}\psi \in q$ iff $\psi \in q'$, and
- for all $\psi \mathsf{U} \psi' \in \mathrm{cl}(\varphi)$, we have $\psi \mathsf{U} \psi' \in q$ iff $\psi' \in q$ or $\psi \in q$ and $\psi \mathsf{U} \psi' \in q'$.

[2] Considering free variables, even though our runtime policies can only ever be sentences, is necessary, because an SA for a policy φ is inductively defined in terms of SAs for its subformulae (i.e., $\mathcal{A}_\varphi$'s subautomata), some of which may contain free variables.

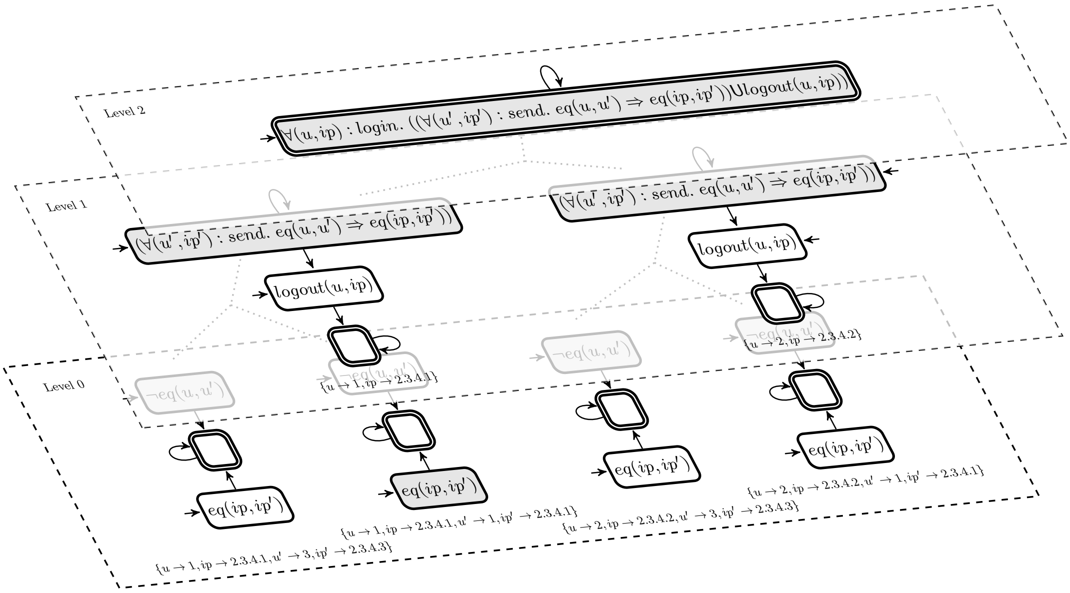

Fig. 1 Spawning on $\{login(1, 2.3.4.1), login(2, 2.3.4.2), send(3, 2.3.4.3), send(1, 2.3.4.1)\}$.

This is similar to the well known syntax directed construction of GBAs (cf. [3]), except that we also need to cater for quantified subformulae. For this purpose, an inductive *spawning function* is defined as follows. If $l > 0$, then $\delta_{\downarrow}(q, (\mathfrak{A}, \sigma))$ yields

$$\left(\bigwedge_{\forall \mathbf{x}:p.\psi \in q} \left(\bigwedge_{(p,\mathbf{d}) \in \sigma} \mathcal{A}_{\psi, v'} \right) \right) \wedge \left(\bigwedge_{\neg \forall \mathbf{x}:p.\psi \in q} \left(\bigvee_{(p,\mathbf{d}) \in \sigma} \mathcal{A}_{\neg\psi, v''} \right) \right),$$

where $v' = v \cup \{\mathbf{x} \mapsto \mathbf{d}\}$ and $v'' = v \cup \{\mathbf{x} \mapsto \mathbf{d}\}$ are sets of valuations, otherwise $\delta_{\downarrow}(q, (\mathfrak{A}, \sigma))$ yields $\top$. Moreover, we set $Q_0 = \{q \in Q \mid \varphi \in q\}$, $\mathcal{F} = \{F_{\psi \mathsf{U} \psi'} \mid \psi \mathsf{U} \psi' \in \mathrm{cl}(\varphi)\}$ with $F_{\psi \mathsf{U} \psi'} = \{q \in Q \mid \psi' \in q \vee \neg(\psi \mathsf{U} \psi') \in q\}$, and $l = \mathrm{depth}(\varphi)$, where $\mathrm{depth}(\varphi)$ is called the *quantifier depth* of φ. For some $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$, $\mathrm{depth}(\varphi) = 0$ iff φ is a quantifier free formula. The remaining cases are inductively defined as follows: $\mathrm{depth}(\forall\, \mathbf{x} : p.\ \psi) = 1 + \mathrm{depth}(\psi)$, $\mathrm{depth}(\psi \wedge \psi') = \mathrm{depth}(\psi \mathsf{U} \psi') = \max(\mathrm{depth}(\psi), \mathrm{depth}(\psi'))$ and $\mathrm{depth}(\neg\varphi) = \mathrm{depth}(\mathsf{X}\varphi) = \mathrm{depth}(\varphi)$.

Lemma 3 *Let $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ (not necessarily a sentence) and v be a valuation. For each accepting run ρ in $\mathcal{A}_{\varphi,v}$ over input $(\overline{\mathfrak{A}}, w)$, $\psi \in \mathrm{cl}(\varphi)$, and $i \geq 0$, we have that $\psi \in \rho(i)$ iff $(\overline{\mathfrak{A}}, w, v, i) \models \psi$.*

Theorem 6 *The constructed SA is correct in the sense that for any sentence $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$, we have that $\mathcal{L}(\mathcal{A}_{\varphi}) = \mathcal{L}(\varphi)$.*

Example 2 Consider the graphical representation of an SA for $\varphi = \mathsf{G}(\forall (u, ip) : login.\ ((\forall (u', ip') : send.\ eq(u, u') \Rightarrow eq(ip, ip'))\mathsf{U} logout(u, ip)))$ in Fig. 1. In a nutshell, φ states that once user u has logged in to the system from IP-address ip, she must not send anything from an IP-address other than ip until logged out. While φ

is not meant to represent a realistic security policy as is, it does help highlight the features of an SA: We first note that level l of $\mathcal{A}_\varphi$ is given by depth$(\varphi) = 2$. As φ is of the form $\mathsf{G}\forall(u, ip) : login.\ \psi$, $\mathcal{A}_\varphi$'s state space is de facto that of an ordinary GBA for an LTL formula of the form $\mathsf{G}p$. Let's now assume that $\sigma = \{login(1, 2.3.4.1), login(2, 2.3.4.2), send(3, 2.3.4.3), send(1, 2.3.4.1)\}$ is an event, which we want $\mathcal{A}_\varphi$ to process. Due to φ's outmost quantifier, the two *login*-actions will lead to the spawning of a conjunction of two subautomata of respective levels $l - 1$ (downward dotted lines). The state space of these subautomata is de facto that of an ordinary GBA for an LTL formula of the form $a\mathsf{U}b$ as one can see in Fig. 1, level 1. These SAs also keep track of a quantified formula, hence the two *send*-actions will also spawn a conjunction of subautomata, basically, to check if $eq(u, u') \Rightarrow eq(ip, ip')$ holds. The respective valuations are given below each SA, whereas the respective current states are marked in grey.

5.2 Monitor construction

Before we look at the actual monitor construction, which is conceptually somewhat similar to [9], let us first introduce some additional concepts and notation: For a finite run ρ in $\mathcal{A}_\varphi$ over $(\overline{\mathfrak{A}}, u)$, we call $\delta_\downarrow(\rho(j), (\mathfrak{A}_j, u_j)) = obl_j$ an *obligation*, where $0 \leq j < |u|$, in that obl_j represents the language to be satisfied after j inputs. That is, obl_j refers to the language represented by the positive Boolean combination of spawned SAs. We say obl_j is *met* by the input, if $(\overline{\mathfrak{A}}^j, u^j) \in \mathrm{good}(obl_j)$ and *violated* if $(\overline{\mathfrak{A}}^j, u^j) \in \mathrm{bad}(obl_j)$. Furthermore, ρ is called *potentially locally accepting*, if it can be extended to a run ρ' over $(\overline{\mathfrak{A}}, u)$ together with some infinite suffix, such that ρ' is locally accepting.

The monitor for a formula $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ can now be described in terms of two mutually recursive algorithms: The main entry point is Algorithm M. It reads an event and issues two calls to a separate Algorithm T: one for φ (under a possibly empty valuation v) and one for $\neg\varphi$ (under a possibly empty valuation v). The purpose of Algorithm T is to detect bad prefixes wrt. the language of its argument formula, call it ψ. It does so by keeping track of those finite runs in $\mathcal{A}_{\psi,v}$ that are potentially locally accepting and where its obligations haven't been detected as violated by the input. If at any time not at least one such run exists, then a bad prefix has been encountered. Algorithm T, in turn, uses Algorithm M to evaluate if obligations of its runs are met or violated by the input observed so far (i.e., it inductively creates submonitors): after the ith input, it instantiates Algorithm M with argument ψ' (under corresponding valuation v') for each $\mathcal{A}_{\psi',v'}$ that occurs in obl_i and forwards to it all observed events from time i on.

Algorithm M (*Monitor*). The algorithm takes a $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ (under a possibly empty valuation v). Its abstract behaviour is as follows: Let us assume an initially empty first-order temporal structure $(\overline{\mathfrak{A}}, u)$. Algorithm M reads an event $(\mathfrak{A}, \sigma)$, prints "$\top$" if $(\overline{\mathfrak{A}}\mathfrak{A}, u\sigma) \in \mathrm{good}(\varphi)$ (resp. "$\perp$" for $\mathrm{bad}(\varphi)$), and returns. Otherwise it prints "?", whereas we now assume that $(\overline{\mathfrak{A}}, u) = (\overline{\mathfrak{A}}\mathfrak{A}, u\sigma)$ holds.[3]

[3] Obviously, the monitor does not really keep $(\overline{\mathfrak{A}}, u)$ around, or it would be necessarily trace-length dependent. $(\overline{\mathfrak{A}}, u)$ is merely used here to explain the inner workings of the monitor.

M1. [Create instances of Algorithm T.] Create two instances of Algorithm T: one with φ and one with $\neg\varphi$, and call them $T_{\varphi,v}$ and $T_{\neg\varphi,v}$, respectively.

M2. [Forward next event.] Wait for next event $(\mathfrak{A}, \sigma)$ and forward it to $T_{\varphi,v}$ and $T_{\neg\varphi,v}$.

M3. [Communicate verdict.] If $T_{\varphi,v}$ sends "no runs", print $\bot$ and return. If $T_{\neg\varphi,v}$ sends "no runs", print $\top$ and return. Otherwise, print "?" and go back to M2. ∎

Algorithm T (*Track runs*). The algorithm takes a $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ (under a corresponding valuation v), for which it creates an SA, $\mathcal{A}_{\varphi,v}$. It then reads an event $(\mathfrak{A}, \sigma)$ and returns if $\mathcal{A}_{\varphi,v}$, after processing $(\mathfrak{A}, \sigma)$, does not have any potentially locally accepting runs, for which obligations haven't been detected as violated. Otherwise, it saves the new state of $\mathcal{A}_{\varphi,v}$, waits for new input, and then checks again, and so forth.

T1. [Create SA.] Create an SA, $\mathcal{A}_{\varphi,v}$.

T2. [Wait for new event.] Let $(\mathfrak{A}, \sigma)$ be the event that was read.

T3. [Update potentially locally accepting runs.] Let B and B' be (initially empty) buffers. If $B = \emptyset$, for each $q \in Q_0$ and for each $q' \in \delta_{\rightarrow}(q, (\mathfrak{A}, \sigma))$: add $(q', [\delta_{\downarrow}(q, (\mathfrak{A}, \sigma))])$ to B. Otherwise, set $B' = B$, and subsequently $B = \emptyset$. Next, for all $(q, [obl_1, \ldots, obl_n]) \in B'$ and for all $q' \in \delta_{\rightarrow}(q, (\mathfrak{A}, \sigma))$: add $(q', [obl_{new}, obl_1, \ldots, obl_n])$ to B, where $obl_{new} = \delta_{\downarrow}(q, (\mathfrak{A}, \sigma))$.

T4. [Create submonitors.] For each $(q, [obl_{new}, obl_1 \ldots, obl_n]) \in B$: call Algorithm M with argument ψ (under corresponding v') for each $\mathcal{A}_{\psi,v'}$ that occurs in obl_{new}.

T5. [Iterate over candidate runs.] Assume $B = \{b_0, \ldots, b_m\}$. Create a counter $j = 0$ and set $(q, [obl_0, \ldots, obl_n]) = b_j$ to be the jth element of B.

T6. [Send, receive, replace.] For all $0 \leq i \leq n$: send $(\mathfrak{A}, \sigma)$ to all submonitors corresponding to SAs occurring in obl_i, and wait for the respective verdicts. For every returned $\top$ (resp. $\bot$) replace the corresponding SA in obl_i with $\top$ (resp. $\bot$).

T7. [Corresponding run has violated obligations?] For all $0 \leq i \leq n$: if $obl_i = \bot$, remove b_j from B and go to T9.

T8. [Obligations met?] For all $0 \leq i \leq n$: if $obl_i = \top$, remove obl_i.

T9. [Next run in buffer.] If $j \leq m$, set j to $j + 1$ and go to step T6.

T10. [Communicate verdict.] If $B = \emptyset$, send "no runs" to the calling Algorithm M and return, otherwise send "some run(s)" and go back to T2. ∎

For a given $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ and $(\overline{\mathfrak{A}}, u)$, let us use $M_{\varphi}(\overline{\mathfrak{A}}, u)$ to denote the successive application of Algorithm M for formula φ, first on u_0, then u_1, and so forth. We then get

Theorem 7 *$M_{\varphi}(\overline{\mathfrak{A}}, u) = \top \Rightarrow (\overline{\mathfrak{A}}, u) \in \mathrm{good}(\varphi)$ (resp. for $\bot$ and $\mathrm{bad}(\varphi)$).*

Example 3 Let's recall the specification used in Example 2, which we are going to use in the following to somewhat exemplify the algorithm's behaviour. Given the previously used event $\sigma = \{login(1, 2.3.4.1), login(2, 2.3.4.2), send(3, 2.3.4.3), send(1, 2.3.4.1)\}$, M_{φ} creates two instances T_{φ} and $T_{\neg\varphi}$, respectively. Note that there is no explicit valuation passed as an argument as φ, being the original user-specification, can be assumed a sentence. We have illustrated this in Fig. 2 (see level 2).

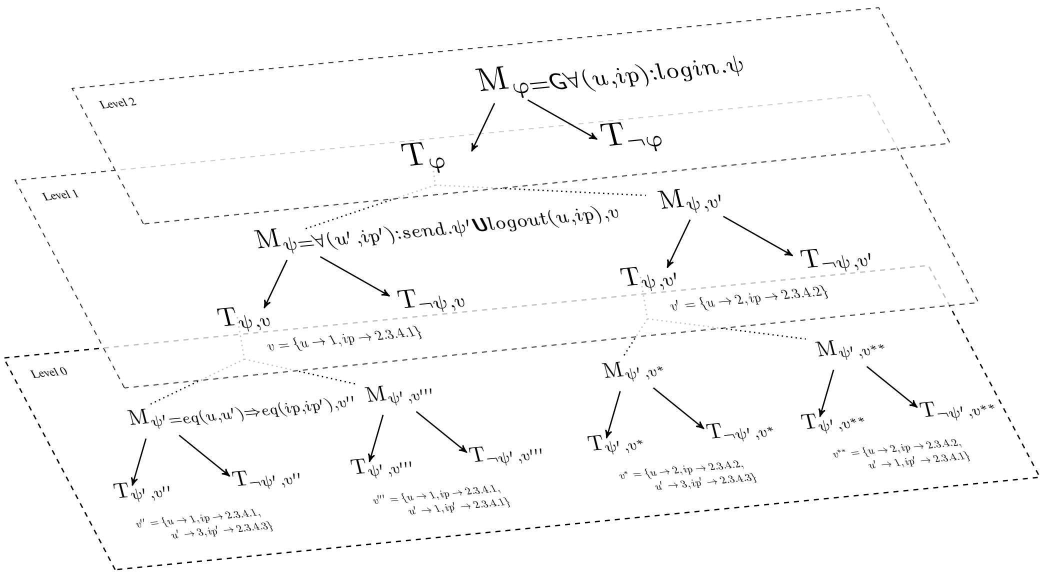

Fig. 2 Monitor processing the event $\{login(1, 2.3.4.1), login(2, 2.3.4.2), send(3, 2.3.4.3), send(1, 2.3.4.1)\}$.

Both T_φ and $T_{\neg\varphi}$ then create their respective SAs, $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$. To be precise, in this very step, they create only the topmost level of respective SAs which, in case of $\mathcal{A}_\varphi$, coincides with level 2 of the SA depicted in Fig. 1, and similarly for $\mathcal{A}_{\neg\varphi}$, although we omit details for brevity. Instead of naively unfolding the respective SAs, Algorithm T has a local buffer B, which keeps track of potentially locally accepting runs; that is, T_φ initialises B with $(q_0, [\delta_\downarrow(q_0, (\mathfrak{A}, \sigma)) = \mathcal{A}_{\psi,v} \wedge \mathcal{A}_{\psi,v'}])$, where q_0 denotes the one and only state of $\mathcal{A}_\varphi$ on level 2 and v, v' are the valuations reflecting the contents of σ. In other words, instead of unfolding $\mathcal{A}_\varphi$ (resp. $\mathcal{A}_{\neg\varphi}$) directly, we store $\delta_\downarrow(q_0, (\sigma, \mathfrak{A}))$ (resp. for $\mathcal{A}_{\neg\varphi}$) in B and then let another monitor deal with its results.

So, as far as T_φ (resp. $T_{\neg\varphi}$) is concerned, it treats the quantified part of φ (resp. $\neg\varphi$) as a proposition, say, p, whose truth value is determined by some oracle (i.e., other monitor); that is, φ is basically of the form $\mathsf{G}p$, for which a corresponding GBA only has a single looping state over proposition p. Hence, T_φ (resp. $T_{\neg\varphi}$) needs to remember only one potentially locally accepting run inside B—the one that loops over p. From this point of view, the name "potentially locally accepting" seems quite fitting, in that a positive answer by the oracle (i.e., submonitor) will, indeed, confirm that the run is a suitable prefix of a satisfying run, if only it continued like that.

Consequently—in a mutually recursive manner—T_φ then creates monitors, one for $\mathcal{A}_{\psi,v}$ and one for $\mathcal{A}_{\psi,v'}$, which we refer to as $M_{\psi,v}$ and $M_{\psi,v'}$, respectively (see Fig. 2, level 1). To stick with our analogy: these monitors serve as the oracles for the simple $\mathsf{G}p$ automaton. (The same process is happening for $T_{\neg\varphi}$, of course, which we disregard for brevity.) By their recursive definition, these new monitors behave like M_φ, except that they do not start with an empty valuation. Moreover, the mutual recursion comes eventually to an end, once there are no further quantifiers to create new Ts and Ms for. It is therefore obvious that Algorithm M terminates, but not necessarily that the respective buffers are not growing unboundedly with increasing trace lengths (and therefore potentially unbounded number of potentially locally accepting runs). The following discussion shall help illustrate this point.

Let us consider the newly created $T_{\psi,v}$ on level 1, Fig. 2, whose buffer is initially as follows: $(q_0, [\delta_{\downarrow}(q_0, (\mathfrak{A}, \sigma)) = \mathcal{A}_{\psi',v''} \wedge \mathcal{A}_{\psi',v'''}])$, where $\mathcal{A}_{\psi',v''}$ and $\mathcal{A}_{\psi',v'''}$ are a reference to the two leftmost SAs on level 0 in Fig. 1. Again, if we interpret these as propositions, we need two further monitors that yield their truth value, which are $M_{\psi',v''}$ and $M_{\psi',v'''}$, respectively. These, in turn, yield four instances of Algorithm T, $T_{\psi',v''}$, $T_{\neg\psi',v''}$, $T_{\psi',v'''}$, and $T_{\neg\psi',v'''}$, respectively (four leftmost instances on level 0, Fig. 2). But as we have reached the end of our recursion and there are no further quantifiers left, the respective buffers of these four instances will never grow and, in fact, both $T_{\neg\psi',v''}$ and $T_{\neg\psi',v'''}$ will have no locally accepting runs. Therefore, $\delta_{\downarrow}(q_0, (\mathfrak{A}, \sigma)) = \top$, and the buffer becomes $(q_0, [])$. Note that we have, again, omitted some details for the negated specification for brevity. However, the point we would like to make here is that potentially locally accepting runs, stored inside the respective buffers, do not necessarily have to be memorised over the entire lifetime of a monitor and can be removed in a process akin to garbage collection known from modern programming languages.

6 Implementation

Now that we have developed also an intuitive understanding of how our monitoring algorithm works, a potential for optimisation should become more or less obvious. In particular, we can easily exploit the idea that the individual levels of an SA are, but GBAs of effectively propositional LTL formulae and are therefore able to use well-known transformations on automata to reduce the overall state space of our monitor. In what follows, we detail on some of these optimisations and then report on experimental results of their implementation.

6.1 Optimisations

We have seen, in particular, in Example 3 that monitoring a formula, such as $\varphi = \mathsf{G}\forall(u, ip) : login.\ \psi$, corresponds to building a hierarchy of submonitors, one for each quantified subformula (and observed action, naturally). On the highest level of this hierarchy, the corresponding monitor will effectively use two GBAs, one for a formula of the form $\mathsf{G}p$ and one for a formula of the form $\neg(\mathsf{G}p)$, where we use p merely as a reference to the submonitors checking the $\forall(u, ip) : login.\ \psi$ part of the original formula, and so forth. In other words, on the topmost level, we have two GBAs for propositional specifications, $\mathsf{G}p$ and $\neg(\mathsf{G}p)$, and on the next level down, one for ψ and one for $\neg\psi$ (as well as for each observed action), and so forth.

This opens up the door to a multitude of possible automata optimisations, which will also help make our monitor more efficient, and some of which were presented also in the context of runtime verification in [9]. Namely, we first convert the individual GBAs into BAs, using the well-known counting construction (cf. [21]). Let $\widetilde{\mathcal{A}_{\varphi}} = (\Sigma, Q, Q_o, \delta, F)$ denote a complete BA obtained this way for

φ, where Σ corresponds to the propositional alphabet of the automaton (thus, completely ignoring the fact that it is used within the context of an SA), Q is its set of states, $Q_0 \subseteq Q$ a set of initial states, $\delta \subseteq Q \times \Sigma \times 2^Q$ the transition relation and $F \subseteq Q$ the set of final states. Then we turn $\widetilde{\mathcal{A}_\varphi}$ into a nondeterministic finite state automaton (NFA), $\hat{\mathcal{A}_\varphi} = (\Sigma, Q, Q_0, \delta, \hat{F})$, where $\hat{F} \supseteq F$ is the set of states for which there exists a path in $\widetilde{\mathcal{A}_\varphi}$, s.t. a strongly connected component can be reached, which contains at least one state from F. In [9], it was shown that $\mathcal{L}(\hat{\mathcal{A}_\varphi}) = \{u \in \Sigma^* \mid$ there exists a $w \in \Sigma^\omega$ s.t. $uw \in \mathcal{L}(\widetilde{\mathcal{A}_\varphi})\}$. In other words, $\hat{\mathcal{A}_\varphi}$ accepts prefixes of elements in $\mathcal{L}(\widetilde{\mathcal{A}_\varphi})$; that is:

Proposition 1 *Every $u \in \mathcal{L}(\hat{\mathcal{A}_\varphi})$, has a potentially locally accepting run in $\mathcal{A}_\varphi$.*

Proof Follows straight from the definitions.

Needless to say, since $\hat{\mathcal{A}_\varphi}$ is but an ordinary NFA, it can be made deterministic and minimal in a language-preserving manner (cf. [24]). Moreover, we can build the same automaton for $\neg\varphi$, and instead of using two automata in parallel, one for φ and one for $\neg\varphi$, we can build the synchronous product of these two (cf. [24]) and use only this one automaton per submonitor—which on top of it all is minimal and deterministic. Let $\mathcal{P}_\varphi = (\Sigma, Q^\varphi \times Q^{\neg\varphi}, Q_0^\varphi \times Q_0^{\neg\varphi}, \delta')$ be the product automaton obtained this way and δ' defined as expected. Then the following formally sums up and puts forth an argument for soundness of this optimisation.

Proposition 2 *If a run reaches a state $(p, q) \in Q^\varphi \times Q^{\neg\varphi}$, s.t. $q \notin \hat{F}^{\neg\varphi}$, then it is potentially locally accepting only in $\mathcal{A}_\varphi$; if $p \notin \hat{F}^\varphi$, then it is potentially locally accepting only in $\mathcal{A}_{\neg\varphi}$; and if $p \in \hat{F}^\varphi$ and $q \in \hat{F}^{\neg\varphi}$ holds, it is potentially locally accepting in $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$.*

Proof Follows from Proposition 1 and soundness of the product construction.

The concrete changes to our algorithms, resulting from these optimisations, are now relatively straightforward to describe. Algorithm M, instead of creating two instances of Algorithm T, merely creates one and interprets its result accordingly (see below). Since for a given φ, Algorithm M no longer calls Algorithm T for both φ and $\neg\varphi$, Algorithm T does not build $\mathcal{A}_\varphi$ in T1, but $\mathcal{P}_\varphi$. Moreover, it performs subsequent operations on this product automaton instead of $\mathcal{A}_\varphi$. Finally, in T10 and in accordance with Proposition 2, if all runs $(q, [obl_{new}, obl_1 \ldots, obl_n]) \in B$ are s.t. that they are only locally accepting in $\mathcal{A}_{\neg\varphi}$, Algorithm T sends "no runs in $\mathcal{A}_\varphi$"; if they are only locally accepting in $\mathcal{A}_\varphi$, it sends "no runs in $\mathcal{A}_{\neg\varphi}$"; and if neither holds, it sends "runs in $\mathcal{A}_\varphi$ and $\mathcal{A}_\varphi$." Algorithm M, in instruction M3, then prints $\bot$, $\top$, ? in either event, respectively.

6.2 Experimental results

To demonstrate feasibility of our algorithm and to get an intuition on its runtime performance (i.e., space consumption at runtime), we have implemented the above.[4] As far as the implementation of the unoptimised version of our algorithm goes, the only liberty we took in deviating from our description so far is that we

[4] Available as open source Scala project on https://github.com/jckuester/ltlfo2mon

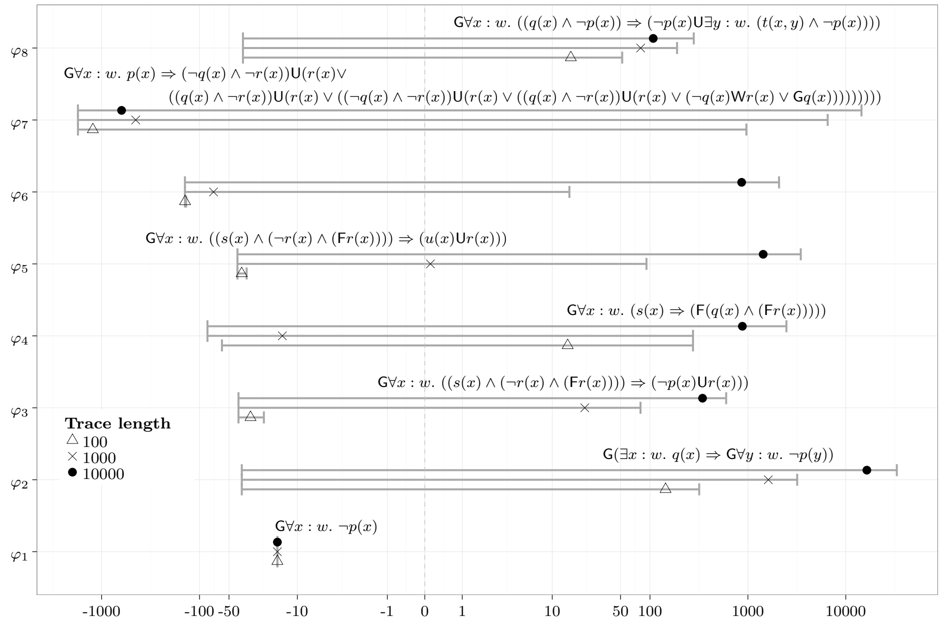

Fig. 3 Difference in space consumption at runtime: SA-based monitor vs. progression.

do not manually construct the GBAs according to the rules laid out in §5.2. We have argued above that the GBAs are basically ordinary propositional automata, hence there is no reason why we cannot employ an "off the shelf" GBA generator, such as lbt[5]. Similarly, for the optimised version, we used the LTL_3-Tools[6] to construct the product automata as described in §6.1. It should be obvious that this does not change any of the results, but instead makes our approach a lot easier to implement. Moreover, our algorithms bear the advantage that it is possible to precompute all the SAs (resp. product automata) that are required at runtime (i.e., we replaced step T1 in Algorithm T with a look-up in a precomputed table of SAs (resp. product automata) and merely use a new valuation each time).

First, we compared our unoptimised implementation with the somewhat naive (but, arguably, easier to implement) approach of monitoring $\mathrm{LTL}^{\mathrm{FO}}$ formulae, described in [7]. There, we used the well-known concept of formula rewriting, sometimes referred to as progression (cf. [2]): a function, P, continuously "rewrites" a formula $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ using an observed event, σ, s.t., $\sigma w \models \varphi \Leftrightarrow w \models P(\varphi, \sigma)$ holds.

As a benchmark for all of our tests, we have used several formulae derived from the well-known specification patterns [17], and added quantification to crucial positions in the formulae.

Some of the results of the comparison between progression and our unoptimised monitor are visualised in Fig. 3. For each $\mathrm{LTL}^{\mathrm{FO}}$ formula corresponding to a pattern, we randomly generated 20 traces of lengths 100, 1000, and 10000, respectively, and passed them to both algorithms. The number of actions per event

[5] http://www.tcs.hut.fi/Software/maria/tools/lbt/
[6] http://ltl3tools.sourceforge.net/

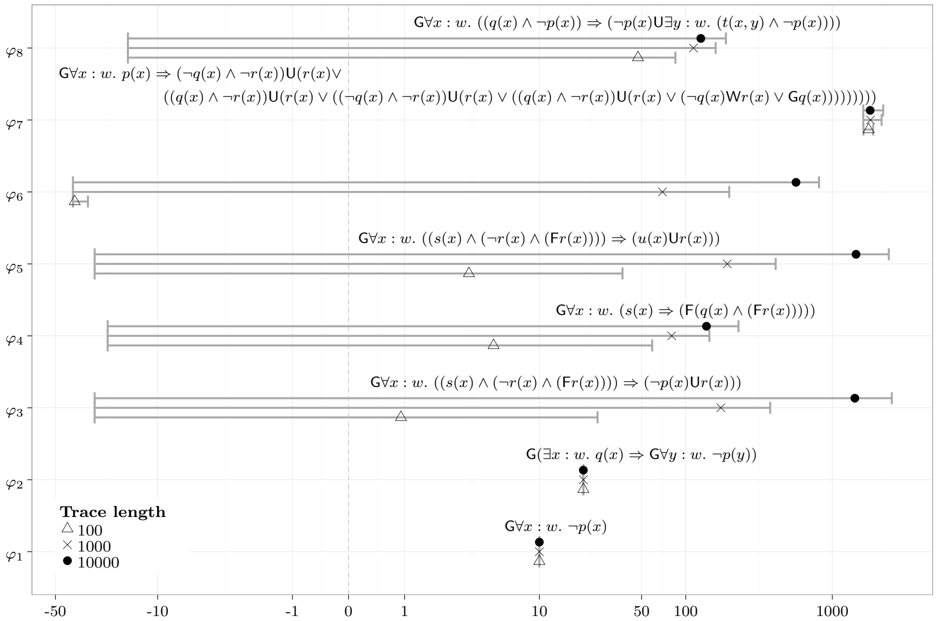

Fig. 4 Difference in space consumption at runtime: optimised vs. unoptimised SA-based monitor.

is uniquely distributed between 0 and 5 and the domain values of ground terms log-normal distributed.[7]

We measured the average space consumption of each algorithm (that is, size of the monitor) at different trace lengths. For progression this is measured simply in terms of the length of the formula at a given time, whereas for the unoptimised SA-based monitor $M_{\varphi,v}$ it is determined recursively as follows: Recall, $M_{\varphi,v}$ first creates two instances of Algorithm T, $T_{\varphi,v}$ and $T_{\neg\varphi,v}$, each of which creates a buffer, call it B_φ, resp. $B_{\neg\varphi}$. Let $B = B_\varphi \cup B_{\neg\varphi}$, and $(q_i, [obl_{i,0}, \ldots, obl_{i,n}])$ be the i-th element of B, then $|M_{\varphi,v}| = \sum_{i=0}^{|B|-1} |(q_i, [obl_{i,0}, \ldots, obl_{i,n}])| = \sum_{i=0}^{|B|-1}(1 + |\widetilde{obl_{i,0}}| + \ldots + |\widetilde{obl_{i,n}}|)$, where $|\widetilde{obl_{i,j}}| = |obl_{i,j}| + \sum_{\mathcal{A}_{\psi,v} \in obl_{i,j}} |M_{\psi,v}|$, i.e., the sum of the top-level monitor's constituents as well as that of all of its submonitors. Finally, we also need to add the total size of the precomputed GBA look-up table. This is measured as $|\mathcal{A}_\varphi| + |\mathcal{A}_{\neg\varphi}| + \sum_{\forall x:p.\psi \in \mathrm{sf}(\varphi)}(|\mathcal{A}_\psi| + |\mathcal{A}_{\neg\psi}|)$, where $|\mathcal{A}_{\psi'}|$ is the size of the obtained GBA when we run lbt on input ψ' with quantified subformulae interpreted as propositions; that is, the sum of the number of its states and transitions as well as the number of literals contained in states. The absolute size of the look-up table for each formula is represented as a black coloured bar in Fig. 5, respectively. We find the smallest look-up table with size 17 for our shortest of all formulae, φ_1, which contains GBAs of the form $\mathsf{G}p$, $\neg\mathsf{G}p$, p and $\neg p$. On the other hand, the

[7] All traces used in this chapter, the definitions of **I**-operators appearing in formulae, as well as the experiments' results in its full extent can be found on https://github.com/jckuester/ltlfo2mon/tree/master/experiments.

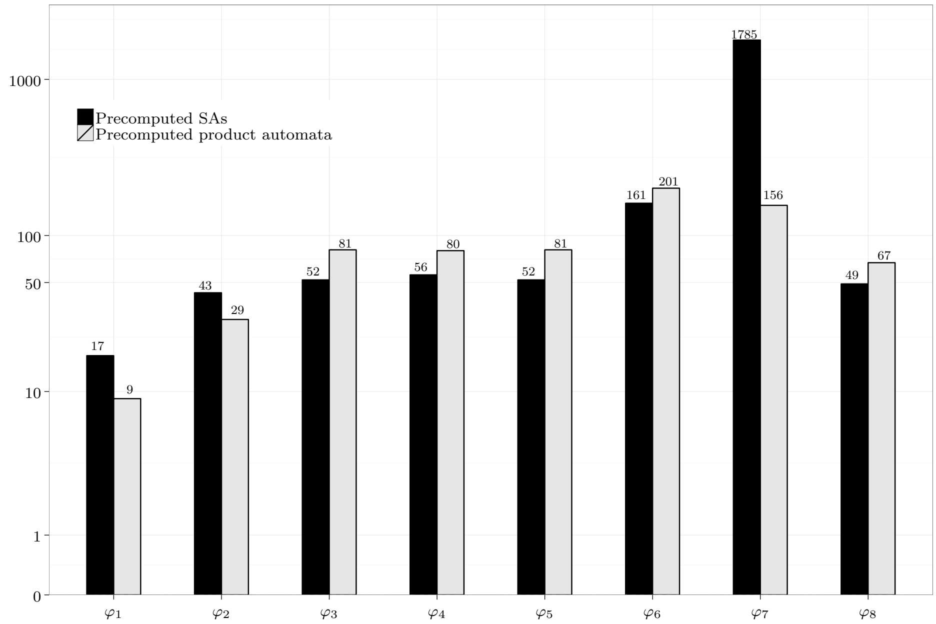

Fig. 5 Size of precomputed look-up tables.

biggest look-up table is the one for φ_7 with size 1785, which is also the longest formula of the patterns used in our experiments.

The end markers on the left of each horizontal bar show how much bigger in the *worst case* an SA-based monitor is for a given formula compared to the corresponding progression-based monitor (and vice versa for the right markers). The small shapes in the middle denote the *average* size difference of the two monitors over the whole length of a trace. This difference is most striking for φ_2 on longer traces (e.g., $\Delta \geq 10000$ for traces of length 10000), where the average almost coincides with the worst case. As such, this example brings to surface one of the potential pitfalls of progression, namely that a lot of redundant information can accumulate over time: If $\exists x : w.\ q(x)$ ever becomes true, then P, which operates purely on a syntactic level, will produce a new conjunct $\mathsf{G}\forall y : w.\ \neg p(y)$ for each new event, even though semantically it is not necessary (or, to use our analogy of before: progression is not very good at garbage collection). Hence, the longer the trace, the greater the average difference in size (similar in case of φ_3 and φ_4).

At first glance, it may seem a curious coincidence that the left markers of each bar align perfectly, because this indicates that for all three traces that belong to a given formula, the SA-based monitor is in the worst case by exactly the same constant k bigger than the progression-based one, irrespective of the trace. However, it makes sense if we consider when this worst case occurs; that is, whenever the SA-based monitor (and consequently also the progression-based one) does not have to memorise any data at all, in which case the size of the SA-based monitor's look-up table weighs the most; that is, the size of the look-up table is almost equal to k. Usually this happens when monitoring commences, hence, there is a perfect alignment on all traces. On the other hand, the worst case for progression arises whenever the amount of data to be memorised by the monitor has reached its maximum. As this depends not only on the formula, but also on the content of the randomly generated traces (and in some of the examples also on their lengths, as seen in the previous example), we generally don't observe alignment on the right.

For those examples that, on average, favour progression, note that the difference in size is less dramatic—a fact, which may be slightly obfuscated by the pseudo-logarithmic scale of the x-axis. Again, the differences in these examples (φ_1, φ_5, φ_6, and φ_7) can be mostly explained by the fact that an SA-based monitor generally wastes more space for "book keeping". Naturally, all examples are also exposed to a degree of randomness due to the generated traces, which would also deserve closer investigation. Hence, these tests are indicative as well as promising, but certainly not yet conclusive.

Second, we ran the optimised version of our monitor described in §6.1 on the same set of formulae and traces as discussed above to hold it against the unoptimised version. Fig. 4 shows the results in the same type of diagram, which we have used to compare ourselves with a progression-based monitor. Again, as we measure the difference between both monitor implementations at runtime, positive values on the x-axis mean that the optimised monitor is smaller in size than the unoptimised one (and vice versa for negative values). One can clearly see that the optimised monitor, in the majority of cases, is on average much smaller than the unoptimised one. However, in the worst case (left markers) it tends to be often bigger than the unoptimised one, as we can observe for formulae φ_3, φ_4, φ_5, φ_6 and φ_8. The perfect alignment of markers on the left, which we have seen before in Fig. 3, again indicate that the worst case occurs independently of the trace-length; that is, the optimised monitor is bigger by a constant k. In this case k correlates exactly with the difference in size of the precomputed look-up table for the optimised and unoptimised monitor, which can be obtained from the numbers given in Fig. 5. Sizes for the GBA look-up tables are shown as black coloured bars and have been discussed above, whereas the look-up tables of product automata are presented in grey. In contrast to former, the latter size is measured as following: $|\mathcal{P}_\varphi| + \sum_{\forall x:p.\psi \in \mathtt{sf}(\varphi)}(|\mathcal{P}_\psi|)$, where $|\mathcal{P}_{\psi'}|$ is the size of the product automaton obtained from the LTL$_3$-Tools on input ψ'. We establish $|\mathcal{P}_{\psi'}|$ equal to $|\mathcal{A}_{\psi'}|$ with the only difference being that $\mathcal{P}_{\psi'}$ is a transition-based automaton, and therefore we count the number of propositional literals labelling transitions instead of states. On the other hand, for formulae φ_1, φ_2 and φ_7, the difference in size is positive, meaning that the optimised monitor is smaller at any point in time during monitoring.

As was the case in our previous comparison, we get the best results when traces are longer; that is, the optimised monitor is considerably smaller on longer traces as the average size and the right markers almost coincide. This suggests that the unoptimised monitor accumulates a lot of data over time, which the optimised one doesn't. This is due to the fact that on the lowest level, the LTL$_3$-Tools generate propositional and therefore minimal finite state machines of which the optimised monitor only needs to store a single state at runtime. This becomes even more apparent when looking at the average number of runs per monitor on the lowest level for the different formulae: for $\varphi_3 = 4$, $\varphi_4 = 5.6$, $\varphi_5 = 4$, $\varphi_6 = 13.8$ and $\varphi_7 = 52.7$.

Finally, let us consider results for φ_1 and φ_2, for which average and worst cases (left and right markers) are identical. We know for these formulae that no "book keeping" is required, and the results show that, indeed, none of our algorithms does. Recall, we have observed the same phenomena in Fig. 3 only for φ_1, as progression accumulates redundant information for φ_2.

7 Towards a hierarchy of effectively monitorable languages

The experimental results of the previous section, although not comprehensive, have demonstrated that one can build an $\mathrm{LTL}^{\mathrm{FO}}$ monitor for a large variety of formulae, such as those on which the popular software specification patterns are based upon; hence, practically useful ones. However, as both the satisfiability as well as the prefix problem are undecidable in our logic (or any other extension of LTL towards full first-order logic), it is natural to ask what properties a language/formula needs to exhibit, s.t. it is effectively monitorable; that is, by a monitor (ours or otherwise) which does not have to store unnecessary or even unbounded amounts of observed data, or by one which does not return ? until the end of time, and so forth.

Due to the undecidability results, there are obviously boundaries to what a monitor can do, irrespective of the algorithm it is based upon. In this section, we will outline and formalise some of these boundaries and thereby provide a first classification towards a kind of hierarchy of monitorable languages or rather: languages, for which practically useful monitors can be built.

Pnueli and Zaks [32] were the first to formalise a notion of *monitorability* in the setting of propositional LTL, which can be concisely expressed in terms of good and bad prefixes as follows.

Definition 10 Some formula $\varphi \in \mathrm{LTL}(\mathrm{AP})$ is *monitorable*, if for all $u \in (2^{\mathrm{AP}})^*$ there exists a $v \in (2^{\mathrm{AP}})^*$ s.t. $uv \in \mathrm{good}(\varphi)$ or $uv \in \mathrm{bad}(\varphi)$ holds.

Conversely, this definition asserts that a monitor (ours or otherwise) cannot sensibly monitor a language that does not have a good or a bad prefix, because these are exactly what the monitor detects. Consider, for example, the propositional LTL formula $\mathsf{G}(r \Rightarrow \mathsf{F}a)$: it describes a typical liveness property, in that always every request is eventually answered some time in the future. This clearly has no good or bad prefix; hence, a monitor adhering to our semantics could only return ? ad infinitum.

For LTL, monitorability can be decided in ExpSpace [5], but the undecidability of the satisfiability (and therefore prefix) problem of $\mathrm{LTL}^{\mathrm{FO}}$ means that for $\mathrm{LTL}^{\mathrm{FO}}$ monitorability is also undecidable by way of a similar reduction already used in the proof of Theorem 5.

However, assuming we have a monitorable $\mathrm{LTL}^{\mathrm{FO}}$ formula in the above sense[8], it does not automatically mean, there exists an efficient monitor for it. Unlike in the setting of propositional LTL, a first-order monitor may have to store parts of or even the entire trace—depending on the formula. For example, an arithmetic

[8] Or rather: one for which monitorability is even obvious as, indeed, there are many, such as the class of all safety and therefore co-safety properties. As a reminder: safety properties are those of which all counterexamples exhibit a bad prefix, whereas a co-safety property is obtained by taking the complement of a safety property (cf. [5]).

LTL$^{\text{FO}}$ formula such as $\mathsf{G}\exists x : p.\ \mathsf{XG}\exists y : q.\ x \neq y$ requires storing all the different elements $p(l)$ occurring in the trace over time. On the other hand, a monitor for a formula, such as $\mathsf{G}\forall x : p.\ x \geq k$ for some constant k, does not need to accumulate *any* information over time at all, as for every new addition to the trace of observations it can be decided whether all occurring $p(l)$ are, s.t. $l \geq k$ holds. Then, if they are, the monitor will continue, and if at least one of them isn't, the monitor has detected a bad prefix and can stop. Since, intuitively, the complexity of monitoring such formulae does not depend on the length of the trace, we refer to them as being *trace-length independent.*

Definition 11 A formula $\varphi \in$ LTL$^{\text{FO}}$ is *trace-length independent*, if there are constants $k, c \in \mathbb{N}$ and a sound and anticipatory monitor for φ, M_φ, s.t. for any trace, where no individual event σ is, s.t. $|\sigma| > c$, we have that $|M_\varphi| \leq k$.

By $|M_\varphi|$, we refer to the size / space consumption of the monitor (see, for example, §6.2).

The constant c serves merely as a bound on the individual events, which in practice always exists, even if one chooses c to be of an extremely large value. Without considering c, however, one could always construct a trace which contains individual events whose cardinality is sufficiently large, s.t. the corresponding monitor's size has to exceed k. Hence, if a k exists, it always exists wrt. a given c.

The reason for demanding a sound and anticipatory monitor for a given φ is due to the fact that one could always come up with a monitor that, say, returns $\perp$ to the user, thus completely ignoring the LTL$^{\text{FO}}$ semantics, but whose space consumption is bounded by 1, for example. But assuming that we are, in fact, dealing with a sound monitor, it could still return ? for all eternity and thus render the monitoring process useless. Hence, we demand that M_φ also be anticipatory, although this constraint (unlike soundeness) could also have been weakened somewhat: all we really need here is that the monitor whenever it reads a good (resp. bad) prefix for φ informs the user about it as soon as possible. In other words, a sound monitor that doesn't return the correct result until some exponential time in the future, could be as useless as the naive, incorrect one we just outlined.

Finally, note that the monitor construction for propositional LTL formulae given in [9] provides a formal argument to our intuition, namely that every monitorable propositional LTL formula is also trace-length independent. Therefore, we have not defined this notion first for LTL and then lifted it to first-order logic, as we have done it earlier in the text, but stated it immediately in the context of LTL$^{\text{FO}}$.

However, Definition 11 is not sufficiently distinctive for our purposes, which becomes obvious when we consider yet another simple (but admittedly practically rather useless) example, the LTL$^{\text{FO}}$ formula $\mathsf{G}(\exists x : p.\ x \geq r \mathsf{U} \exists y : q.\ x = y)$, which helps to illustrate the following point: It is not a priori clear, whether or not a corresponding monitor needs to accumulate trace information over time, because it depends on the actual data inside the trace; that is, if every new event that is added to the trace is s.t. it contains a $p(l)$, s.t. $l \geq r$, where r is some arbitrary constant, and a $q(k)$ with $k = l$, then it is clearly not necessary to keep any of this information around. If, on the other hand, the trace is s.t. it contains different $p(l)$, with $l \geq r$, in each new addition, but no corresponding $q(k)$s (or, at least, for a very long time), then the monitor needs to memorise the individual $p(l)$

until the corresponding $q(k)$ show up. Clearly, if the $q(k)$ never show up, then the monitor will have to remember an unbounded amount of trace information. And if they do show up, its memory consumption increases until that point (after which "garbage collection" will clean up again), which we cannot determine in advance. This is different from the trace-length dependent example given earlier, where the corresponding monitor's space consumption was bound to grow ad infinitum, irrespective of the data in the trace (unless, of course, the monitor finds a bad prefix, in which case it can stop altogether).

Definition 12 Let $|M_\varphi(t)|$ denote the space consumption of monitor M_φ after processing prefix t. Then, φ is called *strongly trace-length dependent*, if it is trace-length dependent and for any such t, there exists a suffix t', s.t. $|M_\varphi(tt')| > |M_\varphi(t)|$, but no suffix t'', s.t. $|M_\varphi(tt'')| < |M_\varphi(t)|$.

To be absolutely clear: the formula $\mathsf{G}(\exists x : p.\ x \geq r \mathsf{U} \exists y : q.\ x = y)$ is merely trace-length dependent, but not strongly trace-length dependent, whereas $\mathsf{G}\exists x : p.\ \mathsf{XG}\exists y : q.\ x \neq y$ is strongly trace-length dependent. Clearly, one can monitor the former, but not the latter, or at least, not for a very long time, because then the monitor will necessarily run out of space. Both formulae, however, satisfy the conditions of monitorability (see Definition 10).

Open problems. Unfortunately, it is currently not known whether these notions of trace-length dependence are decidable, although we strongly suspect that they are not. An intuitive argument for this can be given by the theorem of Rice (cf. [24]), which intuitively asserts that nontrivial properties of Turing machines are not decidable, and the ability to simulate computations of Turing machines in first-order logic. However, to formally apply this argument, one would need to establish if our decision problems do, indeed, pose nontrivial properties, which is equivalent to proving the statement as such. Hence, we leave this as open problem for now.

8 Related work

This is by no means the first work to discuss monitoring of first-order specifications. Motivated by checking temporal triggers and temporal constraints, the monitoring problem for different types of first-order logic has been widely studied, e.g., in the database community. In that context, Chomicki [11] presents a method to check for violations of temporal constraints, specified using (metric) past temporal operators. The logic in [11] differs from $\mathrm{LTL}^{\mathrm{FO}}$, in that it allows natural first-order quantification over a single countable and constant domain, whereas quantified variables in $\mathrm{LTL}^{\mathrm{FO}}$ range over elements that occur at the current position of the trace (see also [22, 6]). Presumably, to achieve the same effect, [11] demands that policies are what is called "domain independent", so that all statements refer to known objects. As such, domain independence is a property of the policy and shown to be undecidable. In contrast, one could say that $\mathrm{LTL}^{\mathrm{FO}}$ has a similar notion of domain independence already built-in, because of its quantifier. Like $\mathrm{LTL}^{\mathrm{FO}}$, the logic in [11] is also undecidable; no function symbols are allowed and relations are required to be finite. However, despite the fact that the prefix problem is not phrased as a decision problem, its basic idea is already denoted by Chomicki as the potential constraint satisfaction problem. In particular, he shows that the set of prefixes of models for a given formula is not recursively enumerable. On the other hand, the monitor in [11] does not tackle this problem and instead solves what we have introduced as the word problem, which, unlike the prefix problem, is decidable.

Basin et al. [4] extend Chomicki's monitor towards bounded future operators using the same logic. Furthermore, they allow infinite relations as long as these are representable by automatic structures, i.e., automata models. In this way, they show that the restriction on formulae to be domain independent is no longer necessary. $\mathrm{LTL}^{\mathrm{FO}}$, in comparison, is more general, in that it allows computable relations and functions. On the other hand, $\mathrm{LTL}^{\mathrm{FO}}$ lacks syntax to directly specify metric constraints.

The already cited work of Hallé and Villemaire [22] describes a monitoring algorithm for a logic with quantification identical to ours, but without function symbols or arbitrary computable relations. The resulting monitors are generated "on the fly" by using syntax-directed tableaux. In our approach, however, it is possible to pre-compute the individual BAs for the respective subformulae of a policy/levels of the SA, and thereby bound the complexity of that part of our monitor at runtime by a constant factor.

Sistla and Wolfson [34] also discuss a monitor for database triggers whose conditions are specified in a logic, which uses an assignment quantifier that binds a single value or a relation instance to a global, rigid variable. Their monitor is represented by a graph structure, which is extended by one level for each updated database state, and as such proportional in size to the number of updates.

Finally, there are works dealing with so called parametric monitoring which, although not based on first-order logic, offer support for monitoring traces carrying data (cf. [1,35,10]). The approach followed by [10] is to "slice" a trace according to the parameters occurring in it and then to forward the n (effectively propositional) subtraces to n monitor instances of the same specification; for example, one per logged-in user or per opened file. In [1], a similar technique is applied for matching regular expressions with the program trace, when restricted to the symbols declared in an expression. All approaches allow the user to add variables to a specification, but only [35] offers quantifiers. However, to restrict their scope, they must directly precede a positive so called parameterised proposition, which is ensured by syntactic rules that prohibit arbitrary nesting or use of negation that could otherwise help to get around this constraint. None of the approaches support arbitrary nesting of quantifiers and temporal operators, use of negation, or function symbols to name just some important restrictions. However, on the plus side, one is able to use optimised monitoring techniques, developed in the propositional domain, and apply them—with these restrictions in mind—to traces carrying data. For Java, a widely used such implementation is JavaMOP [25].

More recently, since the conference version of this article got published, some further related works appeared in the literature, two of which are as follows. Medhat et al. [30] present an LTL_4 extension towards first-order temporal logic, where quantifiers can have metric constraints. It differs to $\mathrm{LTL}^{\mathrm{FO}}$ in that nested quantifiers are restricted to appear leftmost in a formula without being mixed with temporal operators. Their monitoring algorithm, much like ours, also constructs submonitors to evaluate quantifiers at runtime inspired by a divide and conquer algorithm. They then present an implementation using MapReduce (cf. [14]), capable to run in parallel on GPUs, which leads to almost negligible monitoring overhead at runtime.

Moreover, [15] introduced an algorithm that supports the monitoring of first-order logic formulae without restricting to a closed-world assumption as we have done. In their work, the authors argue that this allows for a more natural specification of properties in some cases. However, since their proposed logic differentiates between a foreground and a background part, their monitoring solution for a chosen background logic depends on the availability of a corresponding theory solver. In practice, this means that this approach is targetting those domains, where so called SMT-solvers have made an impact in recent years (e.g., using linear integer inequalities, arrays, etc.), whereas ours is general in that regard. Also see, for example, [31] for an overview on techniques and logics currently used in SMT-solving.

9 Conclusions

To the best of our knowledge, our algorithm is the first to devise impartial monitors, i.e., address the prefix problem instead of a (variant of the) word problem, for policies given in an undecidable first-order temporal logic. Moreover, unlike other approaches, such as [34, 22] and even [7], we are even able to precompute most of the state space required at runtime as the different levels of our SAs correspond to standard GBAs that can be generated before monitoring commences. As required, our monitor is monotonic and in principle trace-length independent. Although, we have identified formulae for which a monitor (which is sound and anticipatory) is bound to grow, since it will need to memorise an increasing amount of trace information over time. While our definitions allow for a classification of formulae and languages according to how efficient a monitor we can build for them, we currently do not know whether or not (strong) trace-length dependence is decidable. Having said that, in many practical scenarios the user knows whether or not a formula is monitorable and trace-length independent, s.t. this is not really a show stopper for first-order monitoring per se, but more of theoretical interest.

Given a $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ of which we know that it can be monitored in a trace-length independent manner, our monitor's size at runtime at any given time is bounded by $O(|\sigma|^{\mathrm{depth}(\varphi)} \cdot 2^{|\varphi|})$, where σ is the current input to the monitor: Throughout the $\mathrm{depth}(\varphi)$ levels of the monitor, there are a total of $O(|\sigma|^{\mathrm{depth}(\varphi)})$ submonitors, which are of size $O(2^{|\varphi|})$, respectively. In contrast, the size of a progression-based monitor, even for obviously trace-length independent formulae, such as φ_2 in Fig. 3 is, in the worst case, proportional to the trace length. Essentially this also means that, for a given formula and a trace, a progression-based monitor has to memorise at least as much data as an SA-based one, but never the other way round (although the SA-based monitor may start with a comparably larger overall overhead).

In Table 1 we have summarised the main results of §2–§4, highlighting again the differences of LTL compared to $\mathrm{LTL}^{\mathrm{FO}}$. Note that as far as trace-length dependence goes, for LTL it is always possible to devise a trace-length independent monitor, irrespective of the specification at hand (cf. [9]).

Table 1 Overview of complexity results.

	LTL	LTL^{FO}
Satisfiability	PSpace-complete	Undecidable
Word problem	< Bilinear-time	PSpace-complete
Model checking	PSpace-complete	ExpSpace-membership, PSpace-hard
Prefix problem	PSpace-complete	Undecidable
Monitorability	ExpSpace-membership, PSpace-hard	Undecidable
Trace-length independence	(By definition)	(Open question)

Acknowledgements. Our thanks go to Patrik Haslum, Michael Norrish and Peter Baumgartner for helpful comments on earlier drafts of this paper.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Proc. 20th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 345–364. ACM (2005)
2. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Annals of Mathematics and Artificial Intelligence **22**, 5–27 (1998). DOI 10.1023/A:1018985923441. URL http://portal.acm.org/citation.cfm?id=590220.590230
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
4. Basin, D., Klaedtke, F., Müller, S.: Policy monitoring in first-order temporal logic. In: Proc. 22nd Intl. Conf. on Computer Aided Verification (CAV), *LNCS*, vol. 6174, pp. 1–18. Springer (2010)
5. Bauer, A.: Monitorability of ω-regular languages. Computing Research Repository (CoRR/arXive) abs/1006.3638, ACM (2010)
6. Bauer, A., Gore, R., Tiu, A.: A first-order policy language for history-based transaction monitoring. In: Proc. 6th Intl. Colloq. on Theoretical Aspects of Computing (ICTAC), *LNCS*, vol. 5684, pp. 96–111. Springer (2009)
7. Bauer, A., Küster, J.C., Vegliach, G.: Runtime verification meets Android security. In: Proc. 4th NASA Formal Methods Symp. (NFM), *LNCS*, vol. 7226, pp. 174–180. Springer (2012)
8. Bauer, A., Küster, J.C., Vegliach, G.: From propositional to first-order monitoring. In: Proc. 4th Intl. Conf. on Runtime Verification (RV), *LNCS*, vol. 8174, pp. 59–75. Springer (2013)
9. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology **20**(4), 14 (2011)
10. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Proc. 15th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *LNCS*, vol. 5505, pp. 246–261. Springer (2009)
11. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. **20**(2), 149–186 (1995)
12. Chomicki, J., Niwinski, D.: On the feasibility of checking temporal integrity constraints. J. Comput. Syst. Sci. **51**(3), 523–535 (1995)
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
14. Dean, J., Ghemawat, S.: MapReduce: A flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)

15. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. In: Proc. 20th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, *LNCS*, vol. 8413, pp. 341–356. Springer (2014)
16. Dong, W., Leucker, M., Schallhart, C.: Impartial anticipation in runtime-verification. In: Proc. 6th Intl. Symp. on Automated Technology for Verification and Analysis (ATVA), *LNCS*, vol. 5311, pp. 386–396. Springer (2008)
17. Dwyer, M., Avrunin, G., Corbett, J.: Patterns in property specifications for finite-state verification. In: Proc. 21st Intl. Conf. on Softw. Eng. (ICSE), pp. 411–420. IEEE (1999)
18. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: Proc. 15th Intl. Conf. on Computer Aided Verification (CAV), *LNCS*, vol. 2725, pp. 27–39. Springer (2003)
19. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
20. Genon, A., Massart, T., Meuter, C.: Monitoring distributed controllers: When an efficient LTL algorithm on sequences is needed to model-check traces. In: Proc. 14th Intl. Symp. on Formal Methods (FM), *LNCS*, vol. 4085, pp. 557–572. Springer (2006)
21. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proc. 15th IFIP WG6.1 Intl. Symp. on Protocol Specification, Testing and Verification XV (IFIP), pp. 3–18. Chapman & Hall (1996)
22. Halle, S., Villemaire, R.: Runtime monitoring of message-based workflows with data. In: Proc. 12th Enterprise Distr. Object Comp. Conf. (EDOC), pp. 63–72. IEEE (2008). DOI 10.1109/EDOC.2008.32. URL http://dx.doi.org/10.1109/EDOC.2008.32
23. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. Software Tools for Technology Transfer **6**(2), 158–173 (2004)
24. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation, first edn. Addison-Wesley (1979)
25. Jin, D., Meredith, P.O., Lee, C., Rosu, G.: JavaMOP: Efficient parametric runtime monitoring framework. In: Proc. 34th Intl. Conf. on Softw. Eng. (ICSE), pp. 1427–1430. IEEE (2012)
26. Kuhtz, L., Finkbeiner, B.: Efficient parallel path checking for linear-time temporal logic with past and bounds. Logical Methods in Computer Science **8**(4) (2012)
27. Libkin, L.: Elements Of Finite Model Theory. Springer (2004)
28. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: Proc. 6th Annual ACM Symp. on Principles of Distributed Computing (PODC), pp. 205–205. ACM (1987)
29. Markey, N., Schnoebelen, P.: Model checking a path. In: Proc. 14th Int. Conf. on Concurrency Theory (CONCUR), *LNCS*, vol. 2761, pp. 248–262. Springer (2003)
30. Medhat, R., Joshi, Y., Bonakdarpour, B., Fischmeister, S.: Parallelized runtime verification of first-order LTL specifications. Technical Report CS-2014-11, University of Waterloo (2014)
31. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). J. ACM **53**(6), 937–977 (2006). DOI 10.1145/1217856.1217859. URL http://doi.acm.org/10.1145/1217856.1217859
32. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Proc. 14th Intl. Symp. on Formal Methods (FM), *LNCS*, vol. 4085, pp. 573–586. Springer (2006)
33. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3), 733–749 (1985)
34. Sistla, A.P., Wolfson, O.: Temporal triggers in active databases. IEEE Trans. Knowl. Data Eng. **7**(3), 471–486 (1995)
35. Stolz, V.: Temporal assertions with parametrized propositions. J. Log. Comp. **20**(3), 743–757 (2010)

A Detailed proofs

Lemma 1. Let φ be a sentence in first-order logic, then we can construct a corresponding $\psi \in \text{LTL}^{\text{FO}}$ s.t. φ has a finite model iff ψ is satisfiable.

Proof We construct ψ as follows. We first introduce a new unary **U**-operator d whose arity is τ and that does not appear in φ. We then replace every subformula in φ, which is of the form $\forall x.\ \theta$, with $\forall x : d.\ \theta$ (resp. for $\exists x.\ \theta$). Next, we encode some restrictions on the interpretation of function and predicate symbols:

- For each constant symbol c in φ, we conjoin the obtained ψ with $d(c)$.
- For each function symbol f in φ of arity n, we conjoin the obtained ψ with $\forall x_1 : d.\ \ldots \forall x_n : d.\ d(f(x_1, \ldots, x_n))$.
- For each predicate symbol p in φ of arity n, we conjoin the obtained ψ with $\forall(x_1, \ldots, x_n) : p.\ d(x_1) \wedge \ldots \wedge d(x_n)$.
- We conjoin $\exists x : d.\ d(x)$ to the obtained ψ to ensure that the domain is not empty.

Finally, we fix the arities of symbols in ψ appropriately to one of the following τ, $\tau \times \ldots \times \tau$, $\tau \times \ldots \times \tau \to \tau$.

Obviously, the formula ψ, constructed by the procedure above, is a syntactically correct LTL^{FO} formula. Now, if ψ is satisfiable by some $(\mathfrak{A}', \sigma)$, where $\mathfrak{A}' = (|\mathfrak{A}'|, I')$ and $\sigma \in (\mathfrak{A}')\text{-}\,\text{Ev}$, it is easy to construct a finite model $\mathfrak{A} = (|\mathfrak{A}|, I)$ s.t. $\mathfrak{A} \models \varphi$ holds in the classical sense of first-order logic: set $|\mathfrak{A}| = d^{I'}$, $c^I = c^{I'}$, $f^I = f^{I'}|_{d^{I'} \times \ldots \times d^{I'}}$, $p^I = p^{I'}$, respectively. By an inductive argument one can show that the LTL^{FO}semantics is preserved. The other direction, if φ is finitely satisfiable, is trivial: set $|\mathfrak{A}'| = \tau^{I'} = |\mathfrak{A}|$, $c^{I'} = c^I$, $f^{I'} = f^I$, respectively, and $\sigma = \{(p, \mathbf{e}) \mid \mathbf{e} \in p^I\} \cup \{(d, e) \mid e \in |\mathfrak{A}|\}$. $\square$

Theorem 3. The word problem for LTL^{FO} is PSpace-complete.

Proof To evaluate a formula $\varphi \in \text{LTL}^{\text{FO}}$ over some linear Kripke structure, $\mathcal{K}$, we can basically use the inductive definition of the semantics of LTL^{FO}: If used as a function, starting in the initial state of $\mathcal{K}$, s_0, it evaluates φ in a depth-first manner with the maximal depth bounded by $|\varphi|$.

To show hardness, we reduce the following problem, which is known to be PSpace-complete: Let $F = Q_1 x_1.\ Q_2 x_2.\ \ldots Q_n x_n.\ E(x_1, x_2, \ldots, x_n)$, where $Q \in \{\forall, \exists\}$ and E is a Boolean expression over variables $x_1, x_2, \ldots, x_n$. Does F evaluate to $\top$ (cf. [19])? The reduction of this problem proceeds as follows. We first construct a formula $\varphi \in \text{LTL}^{\text{FO}}$ in prenex normal form,

$$\varphi = Q_1 x_1 : d.\ Q_2 x_2 : d.\ \ldots Q_n x_n : d.\ E(p_{x_1}(x_1), p_{x_2}(x_2), \ldots, p_{x_n}(x_n)).$$

Then, using an **U**-operator p_{x_i} for every variable x_i, we construct a singleton Kripke structure, $\mathcal{K}$, s.t. $\lambda(s_0) = (\mathfrak{A}, \{(d, 0), (d, 1), (p_{x_1}, 1), (p_{x_2}, 1), \ldots, (p_{x_n}, 1)\})$, where $|\mathfrak{A}| = \{0, 1\}$ and I defined accordingly. It can easily be seen that F evaluates to $\top$ iff $\mathcal{K}$ is a model for φ. Moreover, this construction can be obtained in no more than a polynomial number of steps wrt. the size of the input. $\square$

Theorem 4. The model checking problem for LTL^{FO} is in ExpSpace.

Proof For a given $\varphi \in \text{LTL}^{\text{FO}}$ and $(\mathfrak{A})$-Kripke structure $\mathcal{K}$ defined as usual, where $\mathfrak{A} = (|\mathfrak{A}|, I)$, we construct a propositional Kripke structure $\mathcal{K}'$ and $\varphi' \in \text{LTL}$, s.t. $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi)$ iff $\mathcal{L}(\mathcal{K}') \subseteq \mathcal{L}(\varphi')$ holds. Assuming variable names in φ have been adjusted so that each has a unique name, the construction of φ' proceeds as follows.

Wlog. we can assume $|\mathfrak{A}|$ to be a finite set $\{d_0, \ldots, d_n\}$. We first set φ' to φ and extend the corresponding Γ by the constant symbols $c_{d_0}, \ldots, c_{d_n}$, s.t. $c_{d_i}^I = d_i$, respectively; that is, we add the respective interpretations of each c_{d_i} to I. This step obviously does not require more than polynomial space. We then replace all subformulae in φ' of the form $\nu = Q\,\mathbf{x} : p.\ \psi(\mathbf{x})$ exhaustively with the following constructed ψ':

- Set $\psi' = \top$.
- For each state $s \in S$ do the following:
  - Let $T = \{\mathbf{d} \mid \lambda(s) = (\mathfrak{A}', \sigma), \mathfrak{A}' \sim \mathfrak{A} \text{ and } (p, \mathbf{d}) \in \sigma\}$.

– If $Q = \forall$, then

$$\psi' = \psi' \wedge (\tilde{s} \Rightarrow \bigwedge_{\mathbf{d} \in T} \psi(\mathbf{x})[\mathbf{c} / \mathbf{x}]), \text{where } \mathbf{c} \text{ is s.t. } \mathbf{c}^I = \mathbf{d},$$

otherwise

$$\psi' = \psi' \wedge (\tilde{s} \Rightarrow \bigvee_{\mathbf{d} \in T} \psi(\mathbf{x})[\mathbf{c} / \mathbf{x}]), \text{where } \mathbf{c} \text{ is s.t. } \mathbf{c}^I = \mathbf{d},$$

where $\tilde{s}$ is a fresh, unique predicate symbol meant to represent state s.

Then, for all subformulae in φ' of the form $\tilde{s} \Rightarrow \psi$ we do the following:

- For each $r(\mathbf{t})$ occurring in ψ, where $r \in \mathbf{R}$ and $\mathbf{t}$ are terms, let $\mathbf{d} = \mathbf{t}^I$, and replace $r(\mathbf{t})$ by a fresh, unique predicate symbol $r_{\mathbf{d}}$.

It is easy to see that, indeed, φ' is a syntactically correct standard LTL formula, where all quantifiers have been eliminated. In terms of space complexity, note that in the first loop, we replace each quantified formula by an expression at least $|\mathcal{K}|$ times longer than the original quantified formula. In the worst case, the final formula's length will be exponential in the number of quantifiers.

We now define the propositional Kripke structure $\mathcal{K}' = (S', s_0', \lambda', \rightarrow')$ as follows. Let $S' = S$, $s_0' = s_0$, and $\rightarrow' = \rightarrow$. In what follows, let s be a state and $\lambda(s) = ((|\mathfrak{A}|, I), \sigma)$. (Note, this is the labelling function of $\mathcal{K}$.) The alphabet of $\mathcal{K}'$ is given by 2^{AP}, where $\mathrm{AP} = \{r_{\mathbf{d}} \mid r \in \mathbf{R} \text{ and } \mathbf{d} \in |\mathfrak{A}|\} \cup \{\tilde{s} \mid s \in S\}$. Finally, we define the labelling function of $\mathcal{K}'$ as $\lambda'(s) = \{\tilde{s}\} \cup \{r_{\mathbf{d}} \mid r \in \mathbf{R} \text{ and } r^I(\mathbf{d}) \text{ is true}\}$. It is easy to see that, indeed, $\mathcal{K}'$ preserves all the runs possible through $\mathcal{K}$.

One can show by an easy induction on the structure of φ' that, indeed, $\mathcal{L}(\mathcal{K}) \subseteq \mathcal{L}(\varphi)$ iff $\mathcal{L}(\mathcal{K}') \subseteq \mathcal{L}(\varphi')$ holds. □

Lemma 2. Let $\mathfrak{A}$ be a first-order structure and $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$, then $\mathcal{L}(\varphi)_{\mathfrak{A}} = \{(\overline{\mathfrak{A}}, w) \mid \overline{\mathfrak{A}} \sim \mathfrak{A}, w \in \mathrm{Ev}^{\omega}, \text{ and } (\overline{\mathfrak{A}}, w) \models \varphi\}$. Testing if $\mathcal{L}(\varphi)_{\mathfrak{A}} \neq \emptyset$ is generally undecidable.

Proof Let $K = (x_1, y_1), \ldots, (x_k, y_k)$ be an instance of Post's Correspondence Problem over $\Sigma = \{0, 1\}$, where $x_i, y_i \in \Sigma^+$, which is known to be undecidable in this form. Let us now define a formula $\varphi_K = \exists \gamma : z.\ pcp(\gamma)$, a structure $\mathfrak{A} = (\Sigma^+, I)$, s.t. $pcp^I(u) \Leftrightarrow u = x_{i_1} \ldots x_{i_n} = y_{i_1} \ldots y_{i_n}$, where $u \in \Sigma^+$ and pcp is of corresponding arity. Obviously, $pcp^I(u)$ can be computed in finite time for any given u. Let us now show that $\mathcal{L}(\varphi_K)_{\mathfrak{A}} \neq \emptyset$ iff K has a solution.

($\Rightarrow$:) Because $\mathcal{L}(\varphi_K)_{\mathfrak{A}} \neq \emptyset$, let's assume there is a word $u \in \Sigma^+$ st. $(z, u) \in \sigma$ and $(\mathfrak{A}, \sigma) \in \mathcal{L}(\varphi_K)_{\mathfrak{A}}$. By the choice of pcp^I, there exists a sequence of indices, $i_1, \ldots, i_n$, st. $u = x_{i_1} \ldots x_{i_n} = y_{i_1} \ldots y_{i_n}$, i.e., K has a solution.

($\Leftarrow$:) Let's assume K has a solution, i.e., there exists a word $u \in \Sigma^+$ and a sequence of indices, $i_1, \ldots, i_n$, st. $u = x_{i_1} \ldots x_{i_n} = y_{i_1} \ldots y_{i_n}$. We now have to show that $\mathcal{L}(\varphi_K)_{\mathfrak{A}} \neq \emptyset$. For this purpose, set $\sigma = \{(z, u)\}$, then $(\mathfrak{A}, \sigma) \in \mathcal{L}(\varphi_K)_{\mathfrak{A}}$ and, consequently, $\mathcal{L}(\varphi_K)_{\mathfrak{A}} \neq \emptyset$. □

Theorem 5. The prefix problem for $\mathrm{LTL}^{\mathrm{FO}}$ is undecidable.

Proof By way of a similar reduction used in Theorem 1 already, i.e., for any φ, $\mathfrak{A}$, and $\sigma \in \mathrm{Ev}$ we have that $(\mathfrak{A}, \sigma) \in \mathrm{bad}(\mathsf{X}\varphi)$ iff $\mathcal{L}(\varphi)_{\mathfrak{A}} = \emptyset$. The $\Leftarrow$-direction is obvious. For the other direction:

$$\begin{aligned} & (\mathfrak{A}, \sigma) \in \mathrm{bad}(\mathsf{X}\varphi) \\ \Rightarrow\ & \text{for all } \overline{\mathfrak{A}} \sim \mathfrak{A} \text{ and } w \in \mathrm{Ev}^{\omega}, \text{ we have that } (\mathfrak{A}\overline{\mathfrak{A}}, \sigma w) \not\models \mathsf{X}\varphi \\ \Rightarrow\ & \text{for all } \overline{\mathfrak{A}} \sim \mathfrak{A} \text{ and } w \in \mathrm{Ev}^{\omega}, \text{ we have that } (\overline{\mathfrak{A}}, w) \not\models \varphi \\ \Rightarrow\ & \mathcal{L}(\varphi)_{\mathfrak{A}} = \emptyset \text{ (which is generally undecidable by Lemma 2).} \end{aligned}$$

□

Lemma 3. Let $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ (not necessarily a sentence) and v be a valuation. For each accepting run ρ in $\mathcal{A}_{\varphi,v}$ over input $(\overline{\mathfrak{A}}, w)$, $\psi \in \mathrm{cl}(\varphi)$, and $i \geq 0$, we have that $\psi \in \rho(i)$ iff $(\overline{\mathfrak{A}}, w, v, i) \models \psi$.

Proof We proceed by a nested induction on $\mathrm{depth}(\varphi)$ and the structure of $\psi \in \mathrm{cl}(\varphi)$. For the base case let $\mathrm{depth}(\varphi) = 0$: We fix ρ to be an accepting run in $\mathcal{A}_{\varphi,v}$ over $(\overline{\mathfrak{A}}, w)$, and proceed by induction over those formulae $\psi \in \mathrm{cl}(\varphi)$ which are of depth zero (i.e., without quantifiers) since $\mathrm{depth}(\varphi) = 0$. Therefore, this case basically resembles the correctness argument of Büchi automata for propositional LTL (cf. [3, §5]). For an arbitrary $i \geq 0$, we have

- $\psi = r(\mathbf{t})$:

$$\begin{aligned} r(\mathbf{t}) \in \rho(i) &\Leftrightarrow \mathbf{t}^{I_i} \in r^{I_i} \text{(by the definition of } \delta_\rightarrow\text{)}, \\ &\quad \text{where, as before, for any variable } x \text{ in } \mathbf{t}, \text{ by } x^{I_i} \text{ we mean } v(x) \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \models r(\mathbf{t}) \text{ (by the semantics of LTL}^{\mathrm{FO}}\text{)} \end{aligned}$$

- $\psi = p(\mathbf{t})$: analogous to the above.
- $\psi = \neg\psi'$:

$$\begin{aligned} \neg\psi' \in \rho(i) &\Leftrightarrow \psi' \notin \rho(i) \text{ (by the completeness assumption of all } q \in Q\text{)} \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \not\models \psi' \text{ (by induction hypothesis)} \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \models \neg\psi' \text{ (by the semantics of LTL}^{\mathrm{FO}}\text{)} \end{aligned}$$

- $\psi = \psi_1 \wedge \psi_2$:

$$\begin{aligned} \psi_1 \wedge \psi_2 \in \rho(i) &\Leftrightarrow \{\psi_1, \psi_2\} \subseteq \rho(i) \text{ (by the completeness assumption of all } q \in Q\text{)} \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \models \psi_1' \text{ and } (\overline{\mathfrak{A}}, w, v, i) \models \psi_2 \text{ (by induction hypothesis)} \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \models \psi_1 \wedge \psi_2 \text{ (by the semantics of LTL}^{\mathrm{FO}}\text{)} \end{aligned}$$

- $\psi = \mathsf{X}\psi'$:

$$\begin{aligned} \mathsf{X}\psi' \in \rho(i) &\Leftrightarrow \psi' \in \rho(i+1) \text{ (by the definition of } \delta_\rightarrow\text{)} \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i+1) \models \psi' \text{ (by induction hypothesis)} \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \models \mathsf{X}\psi' \text{ (by the semantics of LTL}^{\mathrm{FO}}\text{)} \end{aligned}$$

- $\psi = \psi_1 \mathsf{U} \psi_2$: we first show the $\Rightarrow$-direction. For this, let us first show that there is a $j \geq i$, such that $(\overline{\mathfrak{A}}, w, v, j) \models \psi_2$ holds. For suppose not, then for all $j \geq i$, we have that $(\overline{\mathfrak{A}}, w, v, j) \not\models \psi_2$ and, consequently, by induction hypothesis $\psi_2 \notin \rho(j)$. By definition of $\delta_\rightarrow$, since $\psi_1 \mathsf{U} \psi_2 \in \rho(i)$ and there isn't a j s.t. $\psi_2 \in \rho(j)$, we have that $\psi_1 \mathsf{U} \psi_2 \in \rho(j)$ for all $j \geq 0$. On the other hand, ρ is accepting in $\mathcal{A}_\varphi$, thus there exist infinitely many $j \geq i$, s.t. $\psi_1 \mathsf{U} \psi_2 \notin \rho(j)$ or $\psi_2 \in \rho(j)$ by the definition of the generalised Büchi acceptance condition $\mathcal{F}$, which is a contradiction. Let us, in what follows, fix the smallest such j. We still need to show that for all $i \leq k \leq j$, $(\overline{\mathfrak{A}}, w, v, k) \models \psi_1$ holds. As j is the smallest such j, where $\psi_2 \in \rho(j)$ it follows that $\psi_2 \notin \rho(k)$ for any such k. As $\psi_1 \mathsf{U} \psi_2 \in \rho(i)$, it follows by definition of $\delta_\rightarrow$ that $\psi_1 \in \rho(i)$ and $\psi_1 \mathsf{U} \psi_2 \in \rho(i+1)$. We can then inductively apply this argument to all $i \leq k < j$, such that $\psi_1 \in \rho(k)$ and $\psi_1 \mathsf{U} \psi_2 \in \rho(k+1)$ hold. The statement then follows from the induction hypothesis.

  Let us now focus on the $\Leftarrow$-direction, i.e., suppose $(\overline{\mathfrak{A}}, w, v, i) \models \psi_1 \mathsf{U} \psi_2$ implies that $\psi_1 \mathsf{U} \psi_2 \in \rho(i)$. By assumption, there is a $j \geq i$, such that $(\overline{\mathfrak{A}}, w, v, j) \models \psi_2$ and for all $i \leq k < j$, we have that $(\overline{\mathfrak{A}}, w, v, k) \models \psi_1$. Therefore, by induction hypothesis, $\psi_2 \in \rho(j)$ and $\psi_1 \in \rho(k)$ for all such k. Then, by the completeness assumption of all $q \in Q$, we also get $\psi_1 \mathsf{U} \psi_2 \in p_j$, and if $j = i$, we are done. Otherwise with an inductive argument similar to the previous case on $k = j-1$, $k = j-2$, $\ldots$, $k = i$, we can infer that $\psi_1 \mathsf{U} \psi_2 \in \rho(k)$.

Let $\mathrm{depth}(\varphi) = n > 0$, i.e., we suppose that our claim holds for all formulae with quantifier depth less than n. We continue our proof by structural induction, where the quantifier free cases are almost exactly as above. Therefore, we focus only on the following case.

- $\psi = \forall \mathbf{x} : p.\ \psi'$: for this case, as before with the U-operator, we will first show the $\Rightarrow$-direction, i.e., for all $i \geq 0$ we have $\forall \mathbf{x} : p.\ \psi' \in \rho(i)$ implies $(\overline{\mathfrak{A}}, w, v, i) \models \forall \mathbf{x} : p.\ \psi'$. By the semantics of LTL$^{\mathrm{FO}}$, the latter is equivalent to for all $(p, \mathbf{d}) \in w_i$, $(\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i) \models \psi'$. If there is no $(p, \mathbf{d}) \in w_i$ the statement is vacuously true. Otherwise, there are some actions $(p, \mathbf{d}) \in w_i$ and

$$\delta_\downarrow(\rho(i), (\mathfrak{A}_i, w_i)) = B \wedge \bigwedge_{(p,\mathbf{d}) \in w_i} \mathcal{A}_{\psi', v \cup \{\mathbf{x} \mapsto \mathbf{d}\}},$$

  where B is a Boolean combination of SAs corresponding to the remaining elements in $\rho(i)$. As ρ is accepting in $\mathcal{A}_{\varphi,v}$, there exists a Y_i satisfying $\delta_\downarrow(\rho(i), (\mathfrak{A}_i, w_i))$, s.t. all $\mathcal{A} \in Y_i$ have an accepting run on input $(\overline{\mathfrak{A}}^i, w^i)$. It follows that Y_i contains an automaton $\mathcal{A}_{\psi', v \cup \{\mathbf{x} \mapsto \mathbf{d}\}}$

for each action $(p, \mathbf{d}) \in w_i$ that has an accepting run ρ'. As the respective levels of these automata is $n-1$, we can use the induction hypothesis and note that the following holds true for each of the $\mathcal{A}_{\psi', v \cup \{\mathbf{x} \mapsto \mathbf{d}\}} \in Y_i$:

$$\text{for all: } \nu \in \mathrm{cl}(\psi') \text{ and } l \geq 0, \nu \in \rho'(l) \text{ iff } (\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i+l) \models \nu,$$

We can now set $\nu = \psi'$, respectively, and $l = 0$, from which it follows that $\psi' \in \rho'(0)$ iff $(\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i) \models \psi'$, respectively. As by construction of an SA the initial states of runs contain the formula which the SA represents, we have $\psi' \in \rho'(0)$ and hence $(\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i) \models \psi'$, respectively. As this holds for all $\mathcal{A}_{\psi', v \cup \{\mathbf{x} \mapsto \mathbf{d}\}}$, where $(p, \mathbf{d}) \in w_i$, it follows by semantics of $\mathrm{LTL}^{\mathrm{FO}}$ that $(\overline{\mathfrak{A}}, w, v, i) \models \forall \mathbf{x} : p.\ \psi'$.
Let us now consider the $\Leftarrow$-direction, i.e., $(\overline{\mathfrak{A}}, w, v, i) \models \forall \mathbf{x} : p.\ \psi'$ implies $\forall \mathbf{x} : p.\ \psi' \in \rho(i)$, which we show by contradiction. Suppose $\forall \mathbf{x} : p.\ \psi' \notin \rho(i)$, which implies by the completeness assumption of all $q \in Q$ that $\neg \forall \mathbf{x} : p.\ \psi' \in \rho(i)$ holds. If there is no $(p, \mathbf{d}) \in w_i$, then $\delta_{\downarrow}(\rho(i), (\mathfrak{A}_i, w_i))$ is equivalent to $\perp$ and ρ could not be accepting. Therefore there must be some $(p, \mathbf{d}) \in w_i$, s.t.

$$\delta_{\downarrow}(\rho(i), (\mathfrak{A}_i, w_i)) = B \wedge \bigvee_{(p, \mathbf{d}) \in w_i} \mathcal{A}_{\neg \psi', v \cup \{\mathbf{x} \mapsto \mathbf{d}\}},$$

where B is a Boolean combination of SAs corresponding to the remaining elements in $\rho(i)$. Because ρ is accepting in $\mathcal{A}_{\varphi, v}$, there exists a Y_i, such that $Y_i \models \delta_{\downarrow}(\rho(i), (\mathfrak{A}_i, w_i))$, and there is at least one SA, $\mathcal{A}' = \mathcal{A}_{\neg \psi', v \cup \{\mathbf{x} \mapsto \mathbf{d}\}} \in Y_i$, with corresponding $(p, \mathbf{d}) \in w_i$, s.t. $(\overline{\mathfrak{A}}^i, w^i)$ is accepted by $\mathcal{A}'$ as input; that is, $\mathcal{A}'$ has an accepting run, ρ', on said input. As this automaton's level is $n-1$, we can apply the induction hypothesis and obtain

$$\text{for all: } \nu \in \mathrm{cl}(\neg \psi') \text{ and } l \geq 0, \nu \in \rho'(l) \text{ iff } (\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i+l) \models \nu.$$

We can now set $\nu = \neg \psi'$ and $l = 0$, and since ν belongs to the initial states in accepting runs, we derive $(\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i) \models \neg \psi'$, which is a contradiction to our initial hypothesis. □

Theorem 6. The constructed SA is correct in the sense that for any sentence $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$, we have that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$.

Proof $\subseteq$: Follows from Lemma 3: let ρ be an accepting run over $(\overline{\mathfrak{A}}, w)$ in $\mathcal{A}_\varphi$. By definition of an (accepting) run, $\varphi \in \rho(0)$, and therefore $(\overline{\mathfrak{A}}, w) \in \mathcal{L}(\varphi)$.

$\supseteq$: We show the more general statement: Given a (possibly not closed) formula $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ and valuation v. It holds that $\{(\overline{\mathfrak{A}}, w) \mid (\overline{\mathfrak{A}}, w, v, 0) \models \varphi\} \subseteq \mathcal{L}(\mathcal{A}_{\varphi, v})$. We define for all $i \geq 0$ the set $\rho(i) = \{\psi \in \mathrm{cl}(\varphi) \mid (\overline{\mathfrak{A}}, w, v, i) \models \psi\}$ for some arbitrary but fixed formula $\varphi \in \mathrm{LTL}^{\mathrm{FO}}$ and valuation v, and arbitrary but fixed $(\overline{\mathfrak{A}}, w)$, where $(\overline{\mathfrak{A}}, w, v, 0) \models \varphi$. Let us now show that $\rho = \rho(0)\rho(1)\ldots$ is a well-defined run in $\mathcal{A}_{\varphi, v}$ over $(\overline{\mathfrak{A}}, w)$: Firstly, from the construction of Q, it follows that for all i, $\rho(i) \in Q$. Secondly, since $\varphi \in \mathrm{cl}(\varphi)$ and $(\overline{\mathfrak{A}}, w, v, 0) \models \varphi$, $\rho(0)$ always contains φ. Thirdly, $\rho(i+1) \in \delta_{\rightarrow}(\rho(i), (\mathfrak{A}_i, w_i))$ holds for all i. The latter is the case iff

- for all $\mathsf{X}\psi \in \mathrm{cl}(\varphi)$: $\mathsf{X}\psi \in \rho(i)$ iff $\psi \in \rho(i+1)$, and
- for all $\psi_1 \mathsf{U} \psi_2 \in \mathrm{cl}(\varphi)$: $\psi_1 \mathsf{U} \psi_2 \in \rho(i)$ iff $\psi_2 \in \rho(i)$ or ($\psi_1 \in \rho(1)$ and $\psi_1 \mathsf{U} \psi_2 \in \rho(i+1)$).

The first condition can be shown as follows:

$$\begin{aligned} \mathsf{X}\psi \in \rho(i) &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \models \mathsf{X}\psi \text{ (by definition of } \rho(i)) \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i+1) \models \psi \text{ (by the semantics of LTL}^{\mathrm{FO}}) \\ &\Leftrightarrow \psi \in \rho(i+1) \text{ (by the definition of } \rho(i+1)). \end{aligned}$$

The second can be shown as follows:

$$\begin{aligned} \psi_1 \mathsf{U} \psi_2 \in \rho(i) &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \models \psi_1 \mathsf{U} \psi_2 \text{ (by definition of } \rho(i)) \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \models \psi_2 \vee (\psi_1 \wedge \mathsf{X}(\psi_1 \mathsf{U} \psi_2)) \\ &\Leftrightarrow (\overline{\mathfrak{A}}, w, v, i) \models \psi_2 \text{ or } ((\overline{\mathfrak{A}}, w, v, i) \models \psi_1 \text{ and } (\overline{\mathfrak{A}}, w, v, i+1) \models \psi_1 \mathsf{U} \psi_2) \\ &\Leftrightarrow \psi_2 \in \rho(i) \text{ or } (\psi_1 \in \rho(1) \text{ and } \psi_1 \mathsf{U} \psi_2 \in \rho(i+1)) \text{ (by definition of } \rho). \end{aligned}$$

It remains to show that ρ is also accepting in $\mathcal{A}_{\varphi,v}$. We proceed by induction on depth(φ). In what follows, let depth(φ) = 0, i.e., we are showing local acceptance only. By the definition of acceptance we must have that for all $\psi_1 \mathsf{U} \psi_2 \in \mathrm{cl}(\varphi)$, there exist infinitely many $i \geq 0$, s.t. $\rho(i) \in F_{\psi_1 \mathsf{U} \psi_2}$, where $F_{\psi_1 \mathsf{U} \psi_2} \in \mathcal{F}$. For suppose not, i.e., there are only finitely many such i, then there is a $k \geq 0$, s.t. for all $j \geq k$ we have $\rho(j) \notin F_{\psi_1 \mathsf{U} \psi_2}$ and therefore $\psi_1 \mathsf{U} \psi_2 \in \rho(j)$ and $\psi_2 \notin \rho(j)$ by definition of $F_{\psi_1 \mathsf{U} \psi_2}$. In particular, from $\psi_1 \mathsf{U} \psi_2 \in \rho(k)$ we derive by construction of $\rho(k)$ that there must be some $g \geq k$, s.t. $(\overline{\mathfrak{A}}^g, w^g) \in \mathcal{L}(\psi_2)$ and thus $\psi_2 \in \rho(k)$ with $g \geq k$. Contradiction.

Let us now assume the statement holds for all formulae with depth strictly less than n and assume depth(φ) = n, where $n > 0$. We don't show local acceptance of ρ as it is virtually the same as in the base case, and instead go on to show that for all $i \geq 0$, there is a Y_i, s.t. $Y_i \models \delta_{\downarrow}(\rho(i), (\mathfrak{A}_i, w_i))$ and all $\mathcal{A} \in Y_i$ are accepting $(\overline{\mathfrak{A}}^i, w^i)$. Let us define the following two sets:

$$Y_i^{\forall} = \{\mathcal{A}_{\psi, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}} \mid \forall \mathbf{x} : p.\ \psi \in \rho(i) \text{ and } (p, \mathbf{d}) \in w_i\}$$

and

$$Y_i^{\exists} = \{\mathcal{A}_{\neg\psi, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}} \mid \neg\forall \mathbf{x} : p.\ \psi \in \rho(i), (p, \mathbf{d}) \in w_i, \text{ and } (\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i) \not\models \psi\}.$$

Set $Y_i = Y_i^{\forall} \cup Y_i^{\exists}$, which by construction satisfies $\delta_{\downarrow}(\rho(i), (\mathfrak{A}_i, w_i))$. We still need to show that every automaton in this set accepts $(\overline{\mathfrak{A}}^i, w^i)$. Now for $\mathcal{A}_{\nu, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}} \in Y_i$ we have either $\nu = \psi$ for some $\forall \mathbf{x} : p.\ \psi \in \rho(i)$ and $(p, \mathbf{d}) \in w_i$, or $\nu = \neg\psi$ for some $\neg\forall \mathbf{x} : p.\ \psi \in \rho(i)$ and $(p, \mathbf{d}) \in w_i$ s.t. $(\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i) \not\models \psi$ holds. In either case by definition of $\rho(i)$ and semantics of $\mathrm{LTL}^{\mathrm{FO}}$, it follows that $(\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i) \models \nu$. Since the level of $\mathcal{A}_{\nu, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}}$ is strictly less than n, we can apply the induction hypothesis and construct an accepting run for $(\overline{\mathfrak{A}}^i, w^i)$, where $(\overline{\mathfrak{A}}, w, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}, i) \models \nu$, in $\mathcal{A}_{\nu, v \cup \{\mathbf{x} \mapsto \mathbf{d}\}}$. The statement follows. □

Theorem 7. $M_{\varphi}(\overline{\mathfrak{A}}, u) = \top \Rightarrow (\overline{\mathfrak{A}}, u) \in \mathrm{good}(\varphi)$ (resp. for $\bot$ and bad(φ)).

Proof We prove the more general statement $M_{\varphi,v}(\overline{\mathfrak{A}}, u) = \top \Rightarrow (\overline{\mathfrak{A}}, u) \in \mathrm{good}(\varphi, v)$, where φ possibly has some free variables and v is a valuation, by a nested induction over depth(φ).

- For the base case let depth(φ) = 0, where φ possibly has free variables, $(\overline{\mathfrak{A}}, u)$ be an arbitrary but fixed prefix and v a valuation. Suppose $M_{\varphi,v}(\overline{\mathfrak{A}}, u)$ returns $\top$ after processing $(\overline{\mathfrak{A}}, u)$, but $(\overline{\mathfrak{A}}, u) \notin \mathrm{good}(\varphi, v)$. By M3 and T10, the buffer of $T_{\neg\varphi,v}$ is empty, i.e., $B_{\neg\varphi,v} = \emptyset$. By T3 and because $\mathcal{A}_{\neg\varphi,v}$ has an accepting run ρ over $(\overline{\mathfrak{A}}, u)$ with some suffix, $B_{\neg\varphi,v}$ contains $(\rho(|u|), [\top])$ after processing $(\overline{\mathfrak{A}}, u)$. Furthermore, because $\delta_{\downarrow}$ yields $\top$ for any input iff depth($\neg\varphi$) = 0, no run in the buffer is ever removed in T7. Contradiction.
- Let *depth*(φ) > 0, $(\overline{\mathfrak{A}}, u)$ be an arbitrary but fixed prefix and v a valuation. Under the same assumptions as above, we will reach a contradiction showing that after processing $(\overline{\mathfrak{A}}, u)$, there is a sequence of obligations $(\rho(|u|), [\mathit{obl}_0, \ldots, \mathit{obl}_n])$ in buffer $B_{\neg\varphi,v}$, which corresponds to an accepting run ρ in $\mathcal{A}_{\neg\varphi,v}$ over $(\overline{\mathfrak{A}}, u)$ with some suffix $(\overline{\mathfrak{A}}', w')$. That is, $\mathrm{M}_{\varphi,v}$ cannot return $\top$, after $B_{\neg\varphi,v}$ is empty, and $B_{\neg\varphi,v}$ containing the above mentioned sequence at the same time. By T3, $B_{\neg\varphi,v}$ contains a sequence $(\rho(|u|), [\mathit{obl}_0, \ldots, \mathit{obl}_n])$ that was incrementally created processing $(\overline{\mathfrak{A}}, u)$ wrt. $\delta_{\rightarrow}$, eventually with some obligations removed if they were detected to be met by the input. We now show that this sequence is never removed from the buffer in T7. Suppose the run has been removed, then there was an $\mathit{obl}_j = \delta_{\downarrow}(\rho(j), (\overline{\mathfrak{A}}_j, u_j))$, that is

$$\left(\bigwedge_{\forall\, \mathbf{x}:p.\psi \in \rho(j)} \left(\bigwedge_{(p,\mathbf{d}) \in u_j} \mathcal{A}_{\psi, v'} \right) \right) \wedge \left(\bigwedge_{\neg\forall\, \mathbf{x}:p.\psi \in \rho(j)} \left(\bigvee_{(p,\mathbf{d}) \in u_j} \mathcal{A}_{\neg\psi, v''} \right) \right),$$

  with $v' = v \cup \{\mathbf{x} \mapsto \mathbf{d}\}$ and $v'' = v \cup \{\mathbf{x} \mapsto \mathbf{d}\}$, evaluated to $\bot$ after l steps, with $0 \leq j \leq l < |u|$. That is, at least one submonitor corresponding to an automaton in the second conjunction has returned $\bot$ (or all submonitors corresponding to automata in a disjunction, for which the following argument would be similar). Wlog. let $\forall \mathbf{x} : p.\psi \in \rho(j)$, $(p, \mathbf{d}) \in u_j$, and $\mathrm{M}_{\psi,v'}(\mathfrak{A}_j, \ldots, \mathfrak{A}_l, u_j, \ldots, u_l) = \bot$, i.e., $\mathrm{M}_{\psi,v'}$ is the submonitor corresponding to $\mathcal{A}_{\psi,v'}$. As *level*(ψ) < *level*(φ), from the induction hypothesis follows that

$(\mathfrak{A}_j, \ldots, \mathfrak{A}_l, u_j, \ldots, u_l) \in bad(\psi, v')$, i.e., $(\mathfrak{A}_j, \ldots, \mathfrak{A}_l\overline{\mathfrak{A}}'', u_j, \ldots, u_l w'') \models \psi$ with evaluation v' for any $(\overline{\mathfrak{A}}'', w'')$, and therefore $(\mathfrak{A}_j, \ldots, \mathfrak{A}_l\overline{\mathfrak{A}}'', u_j, \ldots, u_l w'') \models \neg\forall x : p.\psi$ under valuation v. But as ρ over $(\overline{\mathfrak{A}\mathfrak{A}}', uw')$ is an accepting run in $\mathcal{A}_{\neg\varphi,v}$ and $\forall x : p.\psi \in \rho(j)$, it follows that $(\overline{\mathfrak{A}}^j\overline{\mathfrak{A}}', u^j w') \models \forall x : p.\psi$. Now, we choose $(\overline{\mathfrak{A}}'', w'')$ to be $(\overline{\mathfrak{A}}_{l+1}, \ldots, \overline{\mathfrak{A}}_{|u|}\overline{\mathfrak{A}}'$, $u_{l+1}, \ldots, u_{|u|}w')$. Contradiction.
As for our second statement above, it can be shown similar as before. □