Runtime Enforcement of First-order LTL Properties on Data-aware Business Processes

Riccardo De Masellis[1] and Jianwen Su[2]

[1] Sapienza Università di Roma, Italy
demasellis@dis.uniroma1.it
[2] University of California at Santa Barbara, United States
su@cs.ucsb.edu

Abstract. This paper studies the following problem: given a relational data schema, a temporal property over the schema, and a process that modifies the data instances, how can we enforce the property during each step of the process execution? Temporal properties are defined using a first-order future time LTL (FO-LTL) and they are evaluated under finite and fixed domain assumptions. Under such restrictions, existing techniques for monitoring propositional formulas can be used, but they would require exponential space in the size of the domain. Our approach is based on the construction of a first-order automaton that is able to perform the monitoring incrementally and by using exponential space in the size of the property. Technically, we show that our mechanism captures the semantics of FO-LTL on finite but progressing sequences of instances, and it reports satisfaction or dissatisfaction of the property at the earliest possible time.

1 Introduction

A common pattern in computer science is the constantly increasing complexity of systems, therefore a main challenge is to provide formalisms, techniques, and tools that enable the efficient design and execution of correct and well-functioning systems, despite their complexity. Such a challenge is tackled by business process management (BPM) in the context of business processes. When interested in checking the correctness of a process w.r.t. some properties, two orthogonal approaches can be put in place: *(i)* given a dynamic model of the process, checking *offline*, i.e., before the process is executed, whether every possible execution satisfies the properties, or *(ii)* checking *online* (or *at runtime*) if the current execution satisfies the properties. The first problem is generally called *verification*, and model checking [7] has been the major breakthrough, while the second is called *runtime verification* [20], and it occupies the middle-ground between verification and testing. While both of the aforementioned techniques have considered almost entirely propositional properties, in recent years the emerging of data-centric approaches to business process modeling [24,5] pushed the verification and database community to explore more expressive formalisms for specifying the properties of interest [11,19,4,8]. The key idea of data-centric approaches is to elevate design of data to the same level as design of activities and

their control flows, enriching the classical approaches which lack the connection between process and data. Unfortunately, data-centric business process systems are very challenging to be verified. Indeed, on the one hand, languages for specifying properties are very expressive since they merge the capability of querying a rich structure (e.g., relational) with temporal operators, and on the other hand the presence of data makes such systems infinite state. Unless several restrictions to both the language and possible system evolutions are posed [19,4,8], the classical verification techniques, which check offline all possible evolutions of the systems, are, in general, undecidable. In many practical scenarios, however, it is not necessary to verify all possible executions of the system, but just the current one. For instance, a formal description of the system to be analyzed can be missed, e.g., because unknown, or because highly unstructured (hence an offline analysis cannot be performed) but nonetheless, there are underlying data changing and temporal properties over such data should be enforced.

A representative scenario is health care. Governments have general *guidelines* which do not describe a precise process, but rather a set of cases in which some activities have to be performed. As an example, when a head injury happens, the patient may be rushed to a hospital. During the transportation and upon arrival in emergency room, a crude assessment of the vitals are measured, using the Glasgow coma scale (GCS), injury severity score (ISS) and other test results. *Physicians make decisions on diagnostics and treatments based on collected data and protocols.* Treatment protocols are formulated based on analysis of past patient records and change often. For example, a protocol may state that lightly injured (low ISS score) patients over 65 years old with decreasing GCS scores should have additional tests. After analyzing data, the condition in the protocol may be revised to lightly injured patients over 79 with decreasing GCS or increasing heart rate. In such a scenario, *runtime monitoring of data would help physicians to adhere to the protocol and assists their decisions.*

A further motivation for runtime verification is that it monitors a *concrete* execution of the process, while offline verification is performed on a model of the process (not on the process itself) which, in order to achieve decidability of verification or to cope with the state explosion problem, is usually an approximation of the original process (obtained, e.g., through predicate abstraction or bit state hashing techniques), leading to sound but not complete verification procedures.

In this paper, we focus on the support for runtime enforcement of first-order LTL formulas for data-aware process executions and study the problem of how to incrementally evaluate them. Technically, properties are specified in a first-order language extended with (future time) linear-time logic (LTL) operators and we check them on a sequence of relational data instances incrementally.

The paper makes the following technical contributions. We initiate the study on incremental evaluation of first-order temporal properties over data instances evolving over time, by proposing an automata-based approach: we extend the runtime verification technique presented in [3] to a first-order setting by constructing a first-order Büchi automaton. Such an automaton, along with auxiliary data structures evolving together with the data evolution, is able to monitor the property in an incremental fashion and in exponential space in the size of

the property, while using existing propositional techniques would require exponential space in the size of the domain. More generally, this paper provides an alternative way of performing formal verification of artifact-centric models [17,8] and other business process models such as [14,21].

The paper is organized as follows. Section 2 defines our first-order LTL language and its formal semantics; Section 3 illustrates the automata-based approach and the auxiliary data structures needed for the monitoring; Section 4 describes the space and time complexity; Section 5 provides a picture of the related works and Section 6 concludes the paper.

2 First-order LTL

We assume the data schema to be relational. We define the *data schema* as a tuple $\mathcal{S} = (R_1 \dots R_n, \Delta)$ where $R_1 \dots R_n$ are relation symbols with an associated arity and Δ is a *fixed* a-priori and *finite* set of constants. An instance I of $\mathcal{S}$ interprets each relation symbol R_i with arity n as a relation $R_i^I \subseteq \Delta^n$. Values in Δ are interpreted as themselves, blurring the distinctions between constants and values. Following the tradition of artifact-centric models, we use the terms instance, snapshot or interpretation interchangeably. Given a schema $\mathcal{S}$, symbol $\mathcal{I}$ denotes all possible interpretations for $\mathcal{S}$.

The analysis we perform consists in checking temporal properties while data evolve. In particular, we provide the theoretical foundations for building a module that takes as input a first-order temporal property and, each time data changes, it inspects the new instance and checks whether the temporal property is verified, is falsified or neither of the previous. If the property is falsified, the new instance is rejected, and the execution must continue starting from the previous one, i.e., we rollback to the previous state. Otherwise, the new instance is accepted. In such a scenario we need to recognize a violation at the earlier possible time, in order to be sure that before the last update, i.e., from the previous snapshot, a possible execution that satisfies the formula does exist.

We present a first-order LTL language that merges the capabilities of first-order logic for querying the instance with the LTL temporal operators.

Definition 1 (First-order LTL Language $\mathcal{L}$ Syntax). *Given a data schema $\mathcal{S}$, the set of* closed *first-order LTL formulas Φ of language $\mathcal{L}$ are built with the following syntax:*

$$\begin{aligned} \Phi^\ell &:= true \mid Atom \mid \neg\Phi^\ell \mid \Phi_1^\ell \wedge \Phi_2^\ell \mid \forall x.\Phi^\ell \\ \Phi^t &:= \Phi^\ell \mid \mathbf{X}\Phi^t \mid \Phi_1^t \mathbf{U} \Phi_2^t \mid \neg\Phi^t \mid \Phi_1^t \wedge \Phi_2^t \\ \Phi &:= \Phi^t \mid \neg\Phi \mid \forall x.\Phi \end{aligned}$$

where x is a variable symbol and Atom is an atomic first-order formula or atom, i.e., a formula inductively defined as follows: true is an atomic formula; if t_1 and t_2 are constants in Δ or variables, then $t_1 = t_2$ is an atomic formula and if $t_1 \dots t_n$ are constants or variables and R a relation symbol of arity n, then $R(t_1 \dots t_n)$ is an atomic formula. Since Φ is closed, we assume that all variables symbols are in the scope of a quantifier.

We call *local* formulas the set Φ^ℓ because they do not include any temporal operators. In fact, a formula in Φ^ℓ is a first-order formula with equality but no function symbols and, by expressing a local constraint, it can be checked by looking at a single snapshot. The set of formulas in Φ^t are *temporal* formulas, and indeed they include $\mathbf{X}$ and $\mathbf{U}$ logic symbols that are the usual LTL *next* and *until* operators. Satisfiability of temporal formulas cannot be established by looking at a single snapshot only. Notice that formulas in Φ^t do not include any quantifier for variables occurring in the scope of temporal operators. Finally, the set Φ is made up by formulas that have quantifiers for variables that occur in the scope of temporal operators. We call such variables *across-state* variables. Such formulas are hard and costly to be monitored, because, in general, they require the whole history of snapshots seen so far to determine their truth value. Notice also that the scope of the quantifiers is required to be the entire formula. In other words, our language is not full first-order LTL, since we require all across-state variables to appear in the front of the formula. As an example, $\forall x.(R_1(x)\mathbf{U}(\neg\forall y.(R_2(x,y)\wedge\mathbf{X}R_3(y))))$ is not allowed, since variable y is across-states (because in the scope of $\mathbf{X}$) but its quantifier is not in the front.

We define the propositional symbols $\vee$ and $\exists$ as $\Phi_1 \vee \Phi_2 := \neg(\neg\Phi_1 \wedge \neg\Phi_2)$ and $\exists x.\Phi := \neg\forall x.\neg\Phi$ respectively. Moreover the "finally" $\mathbf{F}$ and "globally" $\mathbf{G}$ LTL operators are defined as $\mathbf{F}\Phi := \mathit{true}\mathbf{U}\Phi$; $\mathbf{G}\Phi := \neg\mathbf{F}\Phi$.

Every formula in $\mathcal{L}$ can be translated into an equivalent formula in *prenex normal form*, i.e., with all quantifier in the front. From now on we assume formulas to be in such a form.

Before showing the semantics of $\mathcal{L}$, we need to introduce the notion of assignment. Let I be a first-order interpretation, i.e., a snapshot for $\mathcal{S}$, an *assignment* η is a function that associates to each free variable x a value $\eta(x)$ in Δ. Let η be an assignment, then $\eta_{x/d}$ is the assignment that agrees with η except for the value $d \in \Delta$ that is now assigned to the variable x. We denote by $\Phi[\eta]$ is the formula obtained from Φ by replacing variables symbols with values in η.

Our analysis is performed at runtime, and hence its semantics is based on finite-length executions, also called paths. Such a semantics is defined starting from the usual infinite-length paths semantics, therefore we first show the latter, and then we turn to the former. An infinite *path* is an infinite sequence of snapshots $I_0, I_1 \ldots$, i.e., given a schema $\mathcal{S}$, it is a function $\pi : \mathbb{N} \to \mathcal{I}$ that assigns a snapshot I_i to each time instant $i \in \mathbb{N}$.

Definition 2 (Infinite path $\mathcal{L}$ Semantics). *Given a formula $\Phi \in \mathcal{L}$ over a schema $\mathcal{S}$, an assignment η, a path π and an instant of time i we have that:*

- $(\pi, i, \eta) \models \Phi^\ell$ *iff* $(\pi(i), \eta) \models \Phi^\ell$, *where* $(\pi(i), \eta) \models \Phi^\ell$ *is the usual first-order logic evaluation function;*
- $(\pi, i, \eta) \models \mathbf{X}\Phi^t$ *iff* $(\pi, i+1, \eta) \models \Phi^t$;
- $(\pi, i, \eta) \models \Phi_1^t \mathbf{U} \Phi_2^t$ *iff for some* $j \geq i$ *we have that* $(\pi, j, \eta) \models \Phi_2^t$ *and for all* $i \leq k < j$ *we have that* $(\pi, k, \eta) \models \Phi_1^t$;
- $(\pi, i, \eta) \models \neg\Phi$ *iff* $(\pi, i, \eta) \not\models \Phi$
- $(\pi, i, \eta) \models \Phi_1 \wedge \Phi_2$ *iff* $(\pi, i, \eta) \models \Phi_1$ *and* $(\pi, i, \eta) \models \Phi_2$;
- $(\pi, i, \eta) \models \forall x.\Phi$ *iff for all* $d \in \Delta$ *we have* $(\pi, i, \eta_{x/d}) \models \Phi$;

Further, $(\pi, \eta) \models \Phi$ *iff* $(\pi, 0, \eta) \models \Phi$ *and, since every formula in* $\mathcal{L}$ *has no free variables, we can simply write* $\pi \models \Phi$*.*

Note that when a formula does not contain any temporal operators, i.e., when it is local, its semantics corresponds exactly to the usual first-order semantics. Indeed, in order to evaluate a local formula, we do not need the whole path, but the current snapshot only. Moreover, the domain Δ is the same for each instant of time (see [15] for a dissertation on different semantics for first-order modal logics).

We now turn to the finite-path semantics. Since the first introduction of LTL by Pnueli [25], several different semantics for finite-path LTL have been proposed, see e.g., [23,13,9]. Here we adapt the one in [3] to our first-order setting. Such a semantics is strongly related with the notion of bad prefixes, that has been established in [22]. Given a formula Φ in $\mathcal{L}$, a *bad prefix* for Φ is a finite path such that any infinite extension of it does not satisfy Φ. In other words, no matter the continuation of the prefix, the formula Φ will be always evaluated to false. As an example, safety properties such as "p holds forever" always have a bad prefix that violates them, that is, a finite path containing a state where p does not hold. Analogously, a *good prefix* can be defined as a finite path which, no matter its continuation, it will always satisfy the property Φ. Eventualities, such as "eventually p holds", always have a finite path that satisfies them. Notice that there are several LTL properties that cannot be satisfied nor falsified by any finite trace, e.g., "infinitely often p holds" as any finite path can be extended to an infinite one satisfying the formula as well as falsifying it. Such formulas are called in [3] *non-monitorable*.

Definition 3 (Finite path $\mathcal{L}$ Semantics). *Given a formula* Φ *over a schema* $\mathcal{S}$ *and a finite path of length* k*, written as* $\pi[k]$*, the truth value of a formula* Φ *on* $\pi[k]$*, denoted by* $[\pi[k] \models \Phi]$*, is an element of the set* $\mathbb{B}_3 = \{true, false, ?\}$ *defined as follows:*

- $[\pi[k] \models \Phi] := true$ *iff* $\pi[k]$ *is bad prefix for* $\neg\Phi$*;*
- $[\pi[k] \models \Phi] := false$ *iff* $\pi[k]$ *is bad prefix for* Φ*;*
- $?$ *otherwise.*

Notice that a bad prefix for $\neg\Phi$ is a good prefix for Φ. The core technical issue of our problem can now be re-formulated as recognizing the bad and good prefixes. Indeed, when a new snapshot I_i is presented as input, we have to check if $I_0 \dots I_i$ is a bad, good prefix or neither of the two, i.e., we have to compute the relation $[I_0 \dots I_i \models \Phi]$. In the classical, propositional version, the problem of recognizing a bad prefix for a propositional formula Ψ can be solved by building a so-called *monitor* for Ψ (see, e.g., [3,9]). The procedure is centered on the construction of a Büchi automaton for Ψ (e.g., following the procedure in [1] or [16]) which is an automaton on infinite strings representing the language $L(\Psi)$ whose accepting condition requires that a final state is visited infinitely often.

Given a propositional formula Ψ over a set of atomic propositions AP, the Büchi automaton $\mathcal{A}$ for Ψ is the automaton such that the language it accepts, denoted by $L(\mathcal{A})$, is the language $L(\Psi)$. Technically, a Büchi automaton is a

tuple $\mathcal{A} = (2^{AP}, Q, \delta, Q_0, F)$ where Q is a set of states, $\delta : Q \times 2^{AP} \rightarrow 2^Q$ the (possibly nondeterministic) transition function, Q_0 the set of initial states and F is the set of final states. A *run* of $\mathcal{A}$ on an infinite word $\alpha = a_0, a_1 \ldots$ (or ω-word) is an infinite state sequence $r(0), r(1) \ldots$ where the following holds: *(i)* $r(0) = q_0$ and *(ii)* $r(i) \in \delta(r(i-1), a_i)$ for $i \leq 1$ if $\mathcal{A}$ is nondeterministic or $r(i) = \delta(r(i-1), a_i)$ for $i \leq 1$ if $\mathcal{A}$ is deterministic. An infinite word α is accepted by $\mathcal{A}$ iff there exists a run $r(0), r(1) \ldots$ which visits one of the states in F infinitely often. In other words, α is accepted if the run $r(0), r(1) \ldots$ cycles in a set of states containing a final state.

Given a propositional formula Ψ, the monitor for Ψ is constructed as follows: (1) the automaton for Ψ is generated and (2) states of the automaton that do not satisfy the Büchi condition, i.e., from which a path that leads to a cycle containing an accepting state does not exist, are marked with "bad". The analysis in (2) is called *emptiness check*. To monitor an execution of a system, it is enough to navigate the automaton's transitions while instances are presented as inputs. When a bad state is reached, the last instance must be rejected. In fact from each bad state there is no way to accept any infinite words belonging to $L(\Psi)$, meaning that Ψ is falsified. Notice that the monitor outputs the truth value ? in any other no bad states, because from them there exists a path leading to the acceptance condition, but the formula can still be falsified later on. Indeed, by using the automaton for Ψ we can recognize the bad prefixes only, but not the good ones. To fully capture the three-valued semantics presented before, two automata have to be used: one for Ψ for recognizing the bad prefixes and one for $\neg\Psi$ recognizing the good ones. We propose an approach that is grounded on the aforementioned technique but that introduces some novelties needed when dealing with first-order properties.

Before entering into the details of our methodology, we point out that our first-order formulas can be translated into an equivalent propositional formula. Indeed, given that no function symbols are in the language and we assume finite domain, the first-order syntax is just a shortcut for the propositional one. Given a first-order formula $\Phi \in \mathcal{L}$, we can build an *equivalent* propositional formula.

In what follows, we refer to [28] for the classical, i.e., on infinite paths, LTL propositional semantics and to [3] to the LTL_3 semantics, i.e., on finite propositional paths.

Lemma 1 (Propositionalization). *Let $\mathcal{L}$ be the language defined before, $\mathcal{L}^p$ a propositional LTL language and Δ a* finite *domain. Then we can build a mapping $\mathsf{p} : \mathcal{L} \rightarrow \mathcal{L}^p$ such that, given a formula $\Phi \in \mathcal{L}$, an infinite path π, and a finite path $\pi[k]$:*

- $\pi \models \Phi \equiv \mathsf{p}(\pi) \models \mathsf{p}(\Phi)$;
- $[\pi[k] \models \Phi] \equiv [\mathsf{p}(\pi[k]) \models \mathsf{p}(\Phi)]$

where, with abuse of notation, $\mathsf{p}(\pi)$ is the natural extension of p to paths, i.e., the path obtained from π by applying function p at the first-order interpretation $\pi(i)$ seen as a logic formula[3]*, for each time instant $i \in \mathbb{N}$.*

[3] We can represent an interpretation I as the conjunction of all positive facts $R_i(d_1, \ldots, d_n)$ when $(d_1, \ldots, d_n) \in R_i^I$ and the conjunction of all negative facts

Proof (Sketch). Function p is inductively defined over $\mathcal{L}$ formulas and, as base case, it associates propositional symbols to first-order atoms. When applied to formulas with a universal quantifier, p returns the conjunction of each assignment for the variables. Given that the domain is finite, we get the claim. □

Notice that there are several ways to map atoms to propositional symbols, hence there are several different propositionalization functions, all equivalent modulo propositional symbol renaming. In the rest of the paper we assume to set one, say function p.

Given a formula $\Phi \in \mathcal{L}$, the size of the formula $\mathsf{p}(\Phi)$ is exponential in the number of universal variables, hence, in the worst case, on the length of Φ. More precisely, its size is $|\Delta|^{|\Phi|}$.

3 First-order automaton

Every formula $\Phi \in \mathcal{L}$ can be propositionalized. Therefore it is easy to see that to monitor Φ we could first propositionalize it, obtaining $\mathsf{p}(\Phi)$, and then we could use existing techniques for monitoring propositional formulas. However, building a monitor requires the construction of the Büchi automaton for $\mathsf{p}(\Phi)$ (see, e.g., [1,16] for the actual procedure) that is exponential in the size of the formula, which, in turn, is exponentially bigger than the original Φ. Given that the automaton construction is $c^{|\mathsf{p}(\Phi)|}$, where c is a constant, we obtain an overall space complexity of is $c^{|\Delta|^{|\Phi|}}$, that is, exponential in the size of the domain and double exponential in the size of the formula.

In this Section we illustrate how to monitor a first-order formula in exponential space in the size of the formula. We make use of a first-order automaton plus some data structures. As it will be clear later on, the auxiliary data structures are used to keep track of assignments to variables. The advantage of this methodology is to decouple the cost of building the automaton from the size of the domain.

Given a formula $\Phi \in \mathcal{L}$, in order to build the first-order automaton for Φ, we first drop all quantifiers from Φ, obtaining an open first-order formula $\hat{\Phi}$ and then we build the automaton for $\hat{\Phi}$. Indeed, given that $\hat{\Phi}$ contains no quantifiers, we can consider the atomic formulas of $\hat{\Phi}$ as propositional symbols and use a standard propositional procedure, e.g., the one in [1], for building the automaton for $\hat{\Phi}$. The formal procedure would require to first propositionalize the atoms of $\hat{\Phi}$, then build the automaton, and lastly use p^{-1} to translate back the propositional symbols into first-order formulas. To ease the presentation we skip the propositionalization step. Indeed, given that $\hat{\Phi}$ does not have quantifiers, function p turns out to be a syntactic renaming of atoms (cf. Lemma 1).

The automaton $\mathcal{A}(\hat{\Phi})$ is likewise a propositional one, except for its transitions and states that are labeled with *open* first-order formulas.

$\neg R_i(d_1, \ldots, d_n)$ when $(d_1, \ldots, d_n) \notin R_i^I$, for all relation symbol $R_i \in \mathcal{L}$ and tuple $(d_1, \ldots, d_n) \in \Delta^n$.

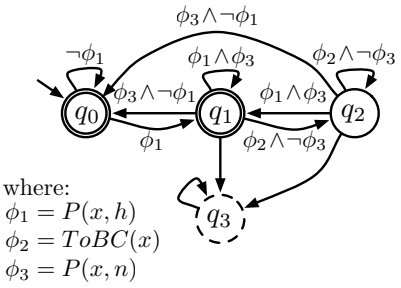

Fig. 1: Graphical representation of the first-order automaton for the formula in Example 1.

Example 1. As an example of the first-order automaton construction, let us consider an hospital that keeps track of the vitals of its patients.

Table P stores information about patients and its attributes are the identifier of the patient and the blood pressure. Moreover, we have a $ToBC$ table with a single attribute storing information about patients which need to be checked by a nurse. We assume possible values for the blood pressure to be normal (n), high (h) or low (l). We want to monitor the property that, for each patient, anytime his blood pressure changes to high, then, in the next instant, a check is needed until his pressure goes back to normal. Such a property can be expressed in $\mathcal{L}$ as:

$$\Phi := \forall x.\mathbf{G}(P(x,h) \rightarrow \mathbf{X}(ToBC(x)\mathbf{U}P(x,n)))$$

The first-order automaton for Φ is graphically represented in Figure 1, and it is obtained by: (1) dropping the quantifiers in the front of the formula getting $\hat{\Phi} = \mathbf{G}(P(x,h) \rightarrow \mathbf{X}(ToBC(x)\mathbf{U}P(x,n)))$ and (2) building the Büchi automaton of $\hat{\Phi}$ according to the procedure in [1] by considering atoms as propositional symbols. Double-circled states q_0 and q_1 are final states, q_0 is the only initial state and the dashed state q_3 is a sink, i.e., *bad* state. To keep the picture readable, labels on transitions to the sink state are omitted, but, intuitively, from each state q_i the bad state is reached whenever any other outgoing transition cannot be executed, hence we have $\delta(q_1, \neg((\neg\phi_1 \wedge \phi_3) \vee (\phi_3 \wedge \neg\phi_1) \vee (\phi_2 \wedge \neg\phi_3)) = \{q_3\}$, $\delta(q_2, \neg(\phi_2 \wedge \neg\phi_3) \vee (\phi_1 \wedge \phi_3) \vee (\phi_3 \wedge \neg\phi_1)) = \{q_3\}$ and $\delta(q_3, true) = \{q_3\}$.

Automaton $\mathcal{A}(\hat{\Phi})$ can be used to recognize bad prefixes of Φ. In general, in order to capture the three-valued semantics in Definition 3, we have to recognize the good prefixes as well, and hence we need the automaton $\mathcal{A}(\neg\hat{\Phi})$. In our example, however, being Φ a safety property, it has no good prefixes because it can be verified by an infinite trace only. The automaton for $\neg\Phi$ is therefore not needed, because it would always return ?. □

Our approach is grounded on the fact that a first-order automaton $\mathcal{A}(\hat{\Phi})$ along with some data structures, is capable of effectively simulating the propositional automata $\mathcal{A}(\mathsf{p}(\Phi))$ needed for recognizing the bad prefixes of $\mathsf{p}(\Phi)$. When using automaton $\mathcal{A}(\neg\hat{\Phi})$ we can recognize also the good prefixes of $\mathsf{p}(\Phi)$. In the rest of the section we are going to show how to use both $\mathcal{A}(\hat{\Phi})$ and $\mathcal{A}(\neg\hat{\Phi})$ for monitoring Φ according to the finite path semantics in Definition 3. To ease the presentation, we proceed in two steps: we first illustrate how, given an assignment η for the variables in Φ, we can recognize the bad prefixes of $\Phi[\eta]$, where $\Phi[\eta]$ denotes the formula obtained from Φ by ignoring the quantifiers and by assigning variables

according to η. We then generalize the procedure by showing how to concurrently monitor all possible assignments using $\mathcal{A}(\hat{\Phi})$ and $\mathcal{A}(\neg\hat{\Phi})$ and how to compose the results obtained for each assignment and for each automata in order to evaluate the original formula.

Let us assume an assignment η for the variables of Φ. We now show a procedure that, given as input: *(i)* a first-order automaton $\mathcal{A}(\hat{\Phi})$; *(ii)* an assignment η for the variables and *(iii)* the snapshots (as data evolve), is able to recognize the bad prefixes for $\mathsf{p}(\Phi[\eta])$ (an analogous procedure can be used on $\mathcal{A}(\neg\hat{\Phi})$ for recognizing the good prefixes). The steps of the procedure follow:

1. we propositionalize the automaton $\mathcal{A}(\hat{\Phi})$ with assignment η. Recalling that a first-order automaton has transitions labeled with first-order formulas, its propositionalization consists in first substituting the variables with values in η and then using function p to obtain propositional formulas in the transitions. We denote such an automaton with $\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])$;
2. We define a marking $M' := Q_0$ as the set of initial states and a marking $M := \emptyset$.
3. At runtime, when a new snapshot I is presented as input:
   (a) $M := M'$ and $M' := \emptyset$;
   (b) for each state $q \in M$ if there exists a transition $(q, \mathsf{p}(\gamma[\eta]), q')$ such that $\mathsf{p}(I) \models \mathsf{p}(\gamma[\eta])$, then $M' := M' \cup q'$.
   (c) we check the emptiness for all $q \in M'$. If at least one state in M' satisfies the Büchi condition, then we return ?, otherwise $false$.

The markings are needed to follow the execution of the snapshots over the automaton which is, in general, nondeterministic, hence more than one state can be contained in the current marking. If from none of the states in the marking it is possible to reach a cycle containing an accepting state, then the sequence of snapshots seen so far is a bad prefix for Φ, and the procedure returns $false$.

By running the same procedure in parallel on the automaton for $\neg\Phi$ (except for step 3(c) where we output *true* if none of the states satisfies the Büchi condition) we recognize also the good prefixes of Φ.

We now prove that this procedure capture exactly the semantics in Definition 3. To this purpose, we reduce to the propositional case where the same result has been proved to hold in [3] and, as a first step, we show that the automaton $\mathsf{p}(\mathcal{A}[\eta])$ and the propositional one for $\mathsf{p}(\Phi[\eta])$ recognize the same language.

Theorem 1. *Given a formula $\Phi \in \mathcal{L}$, let $\hat{\Phi}$ be the (open) formula obtained from Φ by dropping the quantifiers and $\hat{\Phi}[\eta]$ the formula obtained from $\hat{\Phi}$ by substituting all variables with the value given by assignment η. Let moreover:*

- *$\mathcal{A}(\hat{\Phi})$ be the first order automaton for $\hat{\Phi}$;*
- *$\mathcal{A}(\mathsf{p}(\hat{\Phi}[\eta]))$ the propositional automaton for $\mathsf{p}(\hat{\Phi}[\eta])$;*
- *$\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])$ the automaton obtained from $\mathcal{A}(\hat{\Phi})$ by substituting variables with values given by assignment η and by propositionalizing first order formulas with function p;*

then $L(\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])) = L(\mathcal{A}(\mathsf{p}(\hat{\Phi}[\eta])))$.

Proof (Sketch). We prove that the two languages are the same by showing that automata $\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])$ and $\mathcal{A}(\mathsf{p}(\hat{\Phi}[\eta]))$ are the same automaton. As a first step,

since formula $\hat{\Phi}$ has no quantifiers, we abstract from p that trivially associates atoms of the form $R(x, y)$ to propositional symbols R_{x_y}. Then we notice that an assignment η can be viewed as a syntactic manipulation of the formula which changes the name of the variables. Two cases are possible: *(i)* η assigns each variable symbol to a different constant, and *(ii)* η assigns two (or more) variable symbols to the same constant. First case is trivial, while the second one is more involved since two different atoms, e.g., $R(x, y)$ and $R(x, x)$ can become identical, e.g., when $\eta = \{x/a, y/a\}$. If we assume to have two variables only, this requires to prove that $\mathcal{A}(\hat{\Phi}(x, y))|_{y/x}$ and $\mathcal{A}(\hat{\Phi}(x, x))$ are indeed the same automaton, where with $\mathcal{A}(\hat{\Phi}(x, y))|_{y/x}$ we denote the automaton obtained from $\mathcal{A}(\hat{\Phi}(x, y))$ by syntactically replacing each occurrence of y with x. Intuitively, $\mathcal{A}(\hat{\Phi}(x, x))$ shares with $\mathcal{A}(\hat{\Phi}(x, y))$ some states and transitions but it has some less because substituting y to x in Φ may generate contradictions in sub-formulas of Φ (see automaton construction in [1]). Such additional states and transitions, however, are ruled out in $\mathcal{A}(\hat{\Phi}(x, y))|_{y/x}$ after the substitution of y with x. □

Since *(i)* $\mathcal{A}(\mathsf{p}(\phi[\eta]))$ and $\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])$ recognize the same language and *(ii)* the monitoring procedure checks the emptiness per state at each step, we are guaranteed that $\mathsf{p}(\mathcal{A}(\hat{\Phi})[\eta])$ recognizes *minimal* bad prefixes for $\hat{\Phi}[\eta]$.

We now show the key idea for monitoring the whole formula Φ, i.e., all assignment concurrently. Since Φ is in prenex normal form, formula $\mathsf{p}(\Phi)$ has the structure $\bigwedge_{d_1 \in \Delta} \bigvee_{d_2 \in \Delta} \cdots \bigwedge_{d_{n+m} \in \Delta} \mathsf{p}(\Phi[x/d_1, y/d_2 \ldots z/d_{n+m}])$. We can look for the bad (or good) prefixes of such a formula by monitoring $\mathsf{p}(\Phi[x/d_1, y/d_2 \ldots z/d_{n+m}])$ for each assignment $\{x/d_1, y/d_2 \ldots z/d_{n+m}\}$ separately and then composing the results. Indeed, from Definition 3 and the semantics of LTL, it follows that:

- $[\pi[k] \models \phi_1 \wedge \phi_2] = true$ iff $[\pi[k] \models \phi_1] = true \wedge [\pi[k] \models \phi_2] = true$;
- $[\pi[k] \models \phi_1 \wedge \phi_2] = false$ iff $[\pi[k] \models \phi_1] = false \vee [\pi[k] \models \phi_2] = false$;
- $[\pi[k] \models \phi_1 \wedge \phi_2] =?$ otherwise

and analogously for $[\pi[k] \models \phi_1 \vee \phi_2]$.

Given a first order automaton $\mathcal{A}(\hat{\Phi})$, the procedure for recognizing the bad prefixes of Φ is as follows:

1. We define a marking $m : Q \to 2^{\boldsymbol{\eta}}$ (where $\boldsymbol{\eta}$ is the set of possible assignments for Φ) as a function which takes as input a state q of $\mathcal{A}(\hat{\Phi})$ and returns the set of assignments q is marked with; we assign $m'(q) := \boldsymbol{\eta}$ for each $q \in Q_0$ and $m'(q) := \emptyset$ for each $q \notin Q_0$.
2. When a new snapshot I is presented as input:
   (a) $m := m'$ and $m'(q) := \emptyset$ for each $q \in Q$;
   (b) for each state q and for each assignment $\eta \in m(q)$, if there exists a transition (q, γ, q') such that $I \models \gamma[\eta]$ (recall that γ is an open first-order formula) then $m'(q') := m'(q') \cup \eta$;
   (c) for each assignment η we assign a truth value $t(\eta)$ as follows: if there exists at least one state q such that $\eta \in m'(q)$ and the emptiness check from q w.r.t. η (see later) returns true, then $t(\eta) =?$, otherwise $t(\eta) = false$.

(d) recalling that $\Phi := \bigwedge_{d_1 \in \Delta} \bigvee_{d_2 \in \Delta} \cdots \bigwedge_{d_{n+m} \in \Delta} \mathsf{p}(\Phi[x/d_1, y/d_2 \ldots z/d_{n+m}])$, we output the truth value $\bigwedge_{d_1 \in \Delta} \bigvee_{d_2 \in \Delta} \cdots \bigwedge_{d_{n+m} \in \Delta} t(x/d_1, y/d_2 \ldots z/d_{n+m})$.

Notice that the emptiness check is now more complex (step 2(c)), because (unlike the previous case) transitions are first-order formulas. In order to find the minimal bad prefixes, for each q and η such that $\eta \in m'(q)$, we have to substitute values given by η to all transitions involved in paths starting from q. The emptiness check cannot be computed once and for all for a state q because it also depends on the value of η, that is, there can be a transition (q, γ, q') that is first-order satisfiable, i.e., there is at least one assignment that satisfies γ, but $\gamma[\eta]$ is unsatisfiable. We have indeed to be sure that $\mathcal{A}(\hat{\Phi})$ is suitably pruned from unsatisfiable transitions for a given assignment η, before checking for emptiness for q.

By reducing to the previous case, and by running the algorithm in parallel on both $\mathcal{A}(\hat{\Phi})$ and $\mathcal{A}(\neg\hat{\Phi})$ we capture the semantics of Definition 3 for a first-order formula Φ so we recognize minimal good and bad prefixes, i.e., we report a violation or satisfaction at the earliest possible.

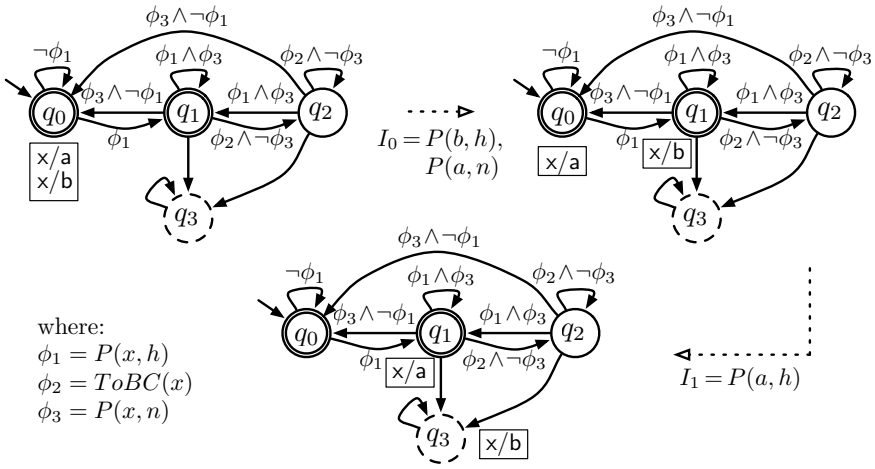

Fig. 2: Graphical representation of the monitoring described in Example 2.

Example 2. We continue Example 1 by showing the evolution of marking m as new instances are presented as inputs. We assume a domain $\Delta = \{a, b\}$. Figure 2 illustrates graphically the assignments $m(q_i)$ inside boxes next to q_i for each $q_i \in Q$. In the initial time instant, according to step 1 of the monitoring procedure, the initial state is marked with all assignments for the free variables, that is, $\eta_1 = x/a$ and $\eta_2 = x/b$. When instance $I_0 = P(b,h), P(a,n)$ is presented as input, we check if it satisfies any of the outgoing transitions from q_0 with substitutions η_1 and η_2 (recall transitions are labeled with open formulas). The automaton has two outgoing transitions from q_0, namely $\delta(q_0, \neg P(x,h)) = \{q_0\}$

and $\delta(q_0, P(x,h)) = \{q_1\}$. Given that $I_0 \models \neg P(x,h)[x/a]$ and $I_0 \models P(x,h)[x/b]$, following step 2(b) of the procedure, the new marking is $m'(q_0) = \{\{x/a\}\}$ and $m'(q_1) = \{\{x/b\}\}$. We then check the emptiness condition for every state marked with an assignment (step 3(c)) and we get $t(x/a) =?$ and $t(x/b) =?$ because both q_0 and q_1 are not bad states and, indeed, from both of them there exists a path leading to a loop containing a final state. Since the original formula is $\bigwedge_{x\in\{a,b\}} \hat{\Phi}(x)$, we get $?\wedge? =?$, meaning that the formula is not yet falsified.

Next instance $I_1 = P(a,h)$ satisfies transitions from q_0 to q_1 with assignment $\{x/a\}$ and from q_1 to q_3 with assignment $\{x/b\}$. Given that q_3 is a bad state, we get $? \wedge false = false$ and the monitoring can be stopped since any prosecution will violate the formula.

4 Time and space complexity

The number of states of the Büchi automaton for a formula Φ is $c^{|\Phi|}$. During the runtime monitoring we keep $|\Delta|^n$ number of assignments where n is the number of variables, hence, in the worst case $|\Delta|^{|\Phi|}$. Given that the automaton is nondeterministic, each state can be marked with all assignments, hence we get a space complexity of $c^{|\Phi|} \cdot |\Delta|^{|\Phi|}$ which is exponential in the size of the formula. For the sake of readability, we presented markings as containing substitutions to both across-state and local variables. Actually, in order to evaluate a formula in $\mathcal{L}$ we do not need to keep assignments for local variables across the execution. Indeed, they are used to check local conditions only. Therefore we can mark states of the automaton with the across-state variables only and use a refined (but trivial) mechanism for computing the markings at each step. Therefore, we get an exponential space approach in the number of across state variables only, which, in many practical cases, is much smaller than the length of the formula. Recall that the naive approach costs $c^{|\Delta|^{|\Phi|}}$ in space.

This gain does not come for free. Indeed, while in the naive approach the emptiness per state can be done offline and once for all after the construction of the automaton, in our case checking the Büchi condition is more involved, because we have to perform it for each assignment separately. As described in the previous Section, we perform such an analysis on the fly. Assuming we use nested depth first search, that is linear time in the size of the automaton, each time a new instance is presented as input, we have to check the emptiness $|\Delta|^{|\Phi|}$ times. We get $|\Delta|^{|\Phi|} \cdot c^{|\Phi|}$ time complexity at each step. We could also check the emptiness for each assignment offline, paying the time cost once for all. This, however, would require to keep, for each state q and for each assignment η information about the badness of q for η, leading to another $c^{|\Phi|} \cdot |\Delta|^{|\Phi|}$ in space. Since the major constraint of this problem is the space rather than the time, we prefer to perform the analysis online.

5 Related Work

Our work stands in the middle-ground between databases and verification, hence it has been influenced by both fields.

Concerning databases, works on incremental evaluation of queries inspired our approach. The work in [12] addresses the problem of efficiently evaluating a datalog query to a database that is being updated. The solution amounts to compute differences between successive database states which, along with the old query answer, can be used to reduce the cost to re-evaluate the query in the new state. Derived (auxiliary) relations are stored to solve the problem. Chomicki [6] focuses on an "history less" approach for checking database integrity temporal constraints. Such constraints are FO-LTL formulas with past-tense operators (*previous time* and *since*). This impacts the verification procedure that is not *runtime* and does not account the undefined ? truth value. An incremental solution, which makes use of auxiliary relations to store intermediate results, is proposed and complexity results are similar. Toman [27] investigated techniques for historical query evaluation in warehouse systems, which, knowing all queries to be asked in advance, physically delete irrelevant past databases. The query language includes temporal variables. Besides, how to evaluate temporal queries over databases has been extensively studied in database literature and ad-hoc query languages has been proposed, such as TSQL2 [26]. Such approaches, however, explicitly represent time in the data, hence they deeply differ from the incremental approach used here. All the aforementioned solutions do not employ automata.

On the other side, the verification community has proposed several techniques for checking dynamic properties of data-centric business processes, all of them based on model checking [7]. Since the presence of data makes the system infinite-state, distinct approaches differ over restrictions used to achieve decidability. In [4] decidability results for verifying temporal properties over data-aware systems are shown, and they are obtained by abstraction and by bounding the size of the so-called *deployed* instances. In [11,19,8] decidability is achieved by constraining the actions that specify how the system evolves. Given that in general several restrictions should be put in place to achieve decidability, our work distances itself from (offline) verification by proposing runtime verification as an alternative to evaluate temporal properties on dynamic systems.

In the runtime verification literature, different formalisms for specifying admissible executions has been proposed, such as ω-regular languages [9], LTL [3] or even μ-calculus [10], but all of them uses propositional languages. In [2] open first-order temporal properties are monitored, and the technique proposed returns assignments that falsify the formula. However, the logic is too expressive for supporting satisfiability and, more important, there is no "lookahead" mechanism of possible future evolutions (automata are not used indeed) so the bad prefixes recognized are not minimal. The work in [18] is the closest to ours, but a naive solution is adopted and no emphasis on complexity is placed.

6 Conclusions and Future Work

This paper initiates the study of runtime monitoring of temporal properties over data evolution using an automata-based technique. We have presented a property specification language $\mathcal{L}$, that consists in a first-order language with LTL operators, fixed domain and quantification across-state. To achieve decidability we constraint the interpretation domain to be finite. From a formula $\Phi \in \mathcal{L}$ we have shown how to build a first-order automaton that, along with auxiliary data structures, can be used for monitoring data evolutions of finite and unknown length. In order to do so, we use the finite path semantics during the runtime evolution of snapshots for recognizing both bad prefixes and good prefixes of a formula. Given that some LTL properties can be violated or satisfied only by infinite paths, such a semantics accounts the truth value ?. The evaluation of the property is based on the traditional emptiness checking and our mechanism captures the semantics of finite paths.

We believe the theoretical complexity results we have obtained justify a deeper investigation of the topic. As a first step in this direction we plan to practically validate our approach. We think that implementing our procedure by making use of symbolic data structures, such as binary decision diagrams, can further improve space and time performances. Besides, an analysis of structural properties of both formulas and automata can reveal ways for implementing optimized data structures for assignments in order to save space. From the theoretical viewpoint, we plan to relax some of the assumptions we have made in this paper, such as the fixed and finite domain assumptions. It would be also of interest to investigate extensions of the temporal component of the language, such as regular expressions, or more powerful logics such as μ-calculus.

Acknowledgments The authors would sincerely like to thank G. De Giacomo, C. Di Ciccio, D. Firmani, F. Leotta and A. Russo for the interesting discussions and suggestions about the paper.

References

1. C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.
2. D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In *CAV*, pages 1–18, 2010.
3. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, Sept. 2011.
4. F. Belardinelli, A. Lomuscio, and F. Patrizi. Verification of deployed artifact systems via data abstraction. In *ICSOC*, pages 142–156, 2011.
5. K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *Proc. 5th Int. Conf. on Business Process Management (BPM)*, Brisbane, Australia, September 2007.
6. J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems*, 20(2):149–186, 1995.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, Cambridge, MA, USA, 1999.

8. E. Damaggio, A. Deutsch, R. Hull, and V. Vianu. Automatic verification of data-centric business processes. In *BPM*, pages 3–16, 2011.
9. M. d'Amorim and G. Rosu. Efficient monitoring of omega-languages. In *CAV*, pages 364–378, 2005.
10. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *TIME*, pages 166–174, 2005.
11. G. De Giacomo, R. De Masellis, and R. Rosati. Verification of conjunctive artifact-centric services. *Int. J. Cooperative Inf. Syst.*, 21(2):111–140, 2012.
12. G. Dong, J. Su, and R. Topor. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14:187–223, 1995.
13. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *CAV*, pages 27–39, 2003.
14. D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Instantaneous soundness checking of industrial business process models. In *BPM*, pages 278–293, 2009.
15. M. Fitting and R. L. Mendelsohn. *First-Order Modal Logic*. Kluwer Academic Press, 1998.
16. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer.
17. C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *Proceedings of 5th International Conference on Service-Oriented Computing (ICSOC)*, Vienna, Austria, September 2007.
18. S. Hallé and R. Villemaire. Runtime monitoring of message-based workflows with data. In *EDOC*, pages 63–72, 2008.
19. B. B. Hariri, D. Calvanese, G. De Giacomo, R. De Masellis, P. Felli, and M. Montali. Verification of description logic knowledge and action bases. In *ECAI*, pages 103–108, 2012.
20. K. Havelund and G. Rosu. Foreword - selected papers from the first international workshop on runtime verification held in paris, july 2001 (rv'01). *Formal Methods in System Design*, 24(2):99–100, 2004.
21. K. Klai, S. Tata, and J. Desel. Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. In *BPM*, pages 294–309, 2009.
22. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
23. Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.
24. A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.
25. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.
26. R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.
27. D. Toman. Expiration of historical databases. In *TIME*, pages 128–135, 2001.
28. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

De Masellis, R., and Su, J. Runtime Enforcement of First-Order LTL Properties on Data-Aware Business Processes. In *Proc. of the 11th Int. Joint Conf. on Service Oriented Computing (ICSOC 2013)*, vol. 8274 of *Lecture Notes in Computer Science*, pp. 54–68. Springer (2013). doi:10.1007/978-3-642-45005-1_5.


```
@inproceedings{DeMasellisS13,
  author    = {Riccardo {De Masellis} and
               Jianwen Su},
  title     = {Runtime Enforcement of First-Order LTL Properties on Data-Aware
               Business Processes},
  booktitle = {ICSOC},
  year      = {2013},
  pages     = {54-68},
  ee        = {http://dx.doi.org/10.1007/978-3-642-45005-1_5}
}
```