Monitoring Data-Aware Business Constraints with Finite State Automata

Riccardo De Masellis
Sapienza Università di Roma, Italy
demasellis@dis.uniroma1.it

Fabrizio M. Maggi
University of Tartu, Estonia
f.m.maggi@ut.ee

Marco Montali
Free University of Bozen-Bolzano, Italy
montali@inf.unibz.it

ABSTRACT

Checking the compliance of a business process execution with respect to a set of regulations is an important issue in several settings. A common way of representing the expected behavior of a process is to describe it as a set of business constraints. Runtime verification and monitoring facilities allow us to continuously determine the state of constraints on the current process execution, and to promptly detect violations at runtime. A plethora of studies has demonstrated that in several settings business constraints can be formalized in terms of temporal logic rules. However, in virtually all existing works the process behavior is mainly modeled in terms of control-flow rules, neglecting the equally important data perspective. In this paper, we overcome this limitation by presenting a novel monitoring approach that tracks streams of process events (that possibly carry data) and verifies if the process execution is compliant with a set of data-aware business constraints, namely constraints not only referring to the temporal evolution of events, but also to the temporal evolution of data. The framework is based on the formal specification of business constraints in terms of first-order linear temporal logic rules. Operationally, these rules are translated into finite state automata for dynamically reasoning on partial, evolving execution traces. We show the versatility of our approach by formalizing (the data-aware extension of) Declare, a declarative, constraint-based process modeling language, and by demonstrating its application on a concrete case dealing with web security.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical logic—*Temporal logic*; H.4.1 [**Information System Applications**]: Office Automation—*Workflow Management*; D.2.3 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; E.0 [**Data**]: General

General Terms

Design, Languages, Management, Verification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSSP '14, May 26-28, 2014, Nanjing, China
Copyright 2014 ACM 978-1-4503-2754-1/14/05 ...$15.00.

Keywords

Compliance Monitoring, Runtime Verification, First-order Linear Temporal Logic, Operational Decision Support

1. INTRODUCTION

Checking the compliance of a business process execution with respect to a set of (dynamic) regulations is an important issue in several settings. A compliance model is, in general, constituted by a set of *business constraints* that can be used to monitor at runtime whether a (running) process instance behaves as expected or not. The declarative nature of business constraints makes it difficult to use procedural languages to describe compliance models [31]. First, the integration of diverse and heterogeneous constraints would quickly make models extremely complex and difficult to maintain. Second, business constraints often target uncontrollable aspects, such as activities carried out by internal autonomous actors or even by external independent entities.

These characteristics make constraint-based approaches suitable for capturing loosely structured and knowledge-intensive processes, such as the treatment of a patient in a hospital. In this case, there is no strict process that physicians have to follow, but only some (underspecified) guidelines and clinical pathways. Even though it is not possible to guarantee that clinical experts behave as specified by the guidelines, it is still crucial to timely detect whether their behavior is aligned with the expected one and, if not, to promptly detect and report deviations. This calls for runtime verification and monitoring as a flexible form of execution support. Since the main focus here is on the dynamics of an evolving system as time flows, LTL (Linear Temporal Logic) has been extensively proposed as a suitable framework for formalizing the properties to be monitored, providing at the same time effective verification procedures. A distinctive feature in the application of LTL to business constraints modeling and monitoring, is that process executions do not continue indefinitely, but eventually reach a termination point. This, in turn, requires to shift from standard LTL over infinite traces (and corresponding techniques based on Büchi automata), to LTL over finite traces (and corresponding techniques based on finite state automata).

In this paper, we represent business constraints using *Declare* [32, 34]. Declare is a declarative language that combines a formal semantics grounded in LTL with a graphical representation for users. Differently from procedural models, which explicitly enumerate the allowed execution traces (considering all the other to be implicitly forbidden), a Declare model describes a process in terms of a set of constraints

that must be satisfied during the process execution. Declare follows a sort of "open world assumption" where every activity can be freely performed unless it is forbidden. One key limitation of Declare is that it completely neglects the data perspective [29], which is crucial in several settings (e.g., healthcare, web security as well as financial institutions that need solid auditing techniques to prevent accounting scandals and frauds). In this light, a compliance model should also include data-aware constraints to guarantee the correct execution of a process in terms of control and data flow. Since, the standard LTL-based formalization of Declare is not sufficient to represent data-aware constraints, it consequently becomes necessary to reformulate them by relying on a more expressive formalism. A first attempt in this direction has been proposed, in [29, 30], where the Event Calculus (EC) is used to formalize data-aware Declare constraints. The limitation of this approach, however, is that it is unable to detect violations at the earliest possible time and also violations that cannot be ascribed to an individual constraint but are determined by the interplay of two or more constraints (cf. [23, 27]). To address this issue, the approach presented in this paper aims at reconciling the data-aware nature of [29, 30] and the advanced reasoning capabilities of [23, 27]. On the one hand, this is done by adopting a (finite-trace variant) of the first-order extension of LTL, which we call FOLTL, to formalize data-aware Declare constraints; on the other hand, a characterization of this logic in terms of finite state automata is exploited to carry out the actual monitoring task. Employing finite state automata for monitoring FOLTL rules allows us to lift the technique used in [23] for the *early detection* of violations to the data-aware case.

The standard FOLTL semantics produces as output a boolean value representing whether the current (finite) trace complies with the monitored constraint or not. In the context of monitoring, however, it is typically required to provide a more fine-grained analysis, which distinguishes between a temporary vs a permanent satisfaction/violation of the constraint, by reasoning on the finite prefixes of the evolving trace [6, 23]. This reflects that, when checking evolving traces, it is not always possible to produce at runtime a definitive answer about compliance. To tackle this issue, we further extend FOLTL for finite traces with the four-valued semantics of RV-FOLTL (FOLTL for Runtime Verification). This makes the approach able to provide advanced diagnostics to the end users, reporting which constraints are violated and *why*. More specifically, our approach does not only communicate if a running trace is currently compliant with the constraint model, but also computes the *state* of each constraint, which is one among *temporarily satisfied*, *permanently satisfied*, *temporarily violated* and *permanently violated*. The first state attests that the monitored process instance is currently compliant with the constraint, but it can violate the constraint in the future. The second state indicates that the constraint is satisfied permanently, i.e., it is no longer possible to violate the constraint. The third state models that the constraint is currently violated, but it is possible to bring it back to a satisfied state by continuing the trace. The last state models the situation where the violation cannot be repaired anymore.

The paper is structured as follows. Section 2 introduces the basic Declare notation and gives an overview on existing analysis tools for Declare. Section 3 recalls FOLTL, and shows how to apply it for formalizing data-aware Declare. In Section 4, the monitoring technique is thoroughly investigated.

Table 1: Graphical notation and LTL formalization of some Declare templates

TEMPLATE	FORMALIZATION
existence(A)	$\Diamond A$
absence(A)	$\neg\Diamond A$
choice(A,B)	$\Diamond A \vee \Diamond B$
exclusive choice(A,B)	$(\Diamond A \vee \Diamond B) \wedge \neg(\Diamond A \wedge \Diamond B)$
responded existence(A,B)	$\Diamond A \rightarrow \Diamond B$
response(A,B)	$\Box(A \rightarrow \Diamond B)$
precedence(A,B)	$\neg B \mathcal{W} A$
alternate response(A,B)	$\Box(A \rightarrow \bigcirc(\neg A \mathcal{U} B))$
alternate precedence(A,B)	$(\neg B \mathcal{W} A) \wedge \Box(B \rightarrow \bigcirc(\neg B \mathcal{W} A))$
chain response(A,B)	$\Box(A \rightarrow \bigcirc B)$
chain precedence(A,B)	$\Box(\bigcirc B \rightarrow A)$
not resp. existence(A,B)	$\Diamond A \rightarrow \neg\Diamond B$
not response(A,B)	$\Box(A \rightarrow \neg\Diamond B)$
not precedence(A,B)	$\Box(A \rightarrow \neg\Diamond B)$
not chain response(A,B)	$\Box(A \rightarrow \neg\bigcirc B)$
not chain precedence(A,B)	$\Box(A \rightarrow \neg\bigcirc B)$

Section 5 grounds our approach on a concrete case study in the context of web security. Section 6 further discusses the concrete of the framework and its versatility. Finally, Section 7 concludes the paper and spells out directions for future work.

2. PRELIMINARIES

In this section, we introduce some background notions about Declare, and give an overview of the existing techniques for the formal analysis and runtime monitoring of Declare models.

2.1 Declare

Declare is a declarative process modeling language originally introduced by Pesic and van der Aalst in [33]. Instead of explicitly specifying the flow of the interactions among process activities, Declare describes a set of constraints that must be satisfied throughout the process execution. The possible orderings of activities are implicitly specified by constraints and anything that does not violate them is possible during execution. In comparison with procedural approaches that produce "closed" models, i.e., all what is not explicitly specified is forbidden, Declare models are "open" and tend to offer more possibilities for the execution. In this way, Declare enjoys flexibility and is very suitable for specifying compliance models that are used to check if the behavior of a system complies with desired regulations. The compliance model defines the rules related to a single process instance, and the overall expectation is that all instances comply with the model.

A Declare model consists of a set of constraints applied to (atomic) activities. Constraints, in turn, are based on templates. Templates are patterns that define parameterized classes of properties, and constraints are their concrete instantiations. Table 1 summarizes some Declare templates (the reader can refer to [33] for a full description of the language), where the $\Diamond$, $\Box$ and $\mathcal{W}$ LTL operators have the following intuitive meaning (see Section 3.1 for the formal

semantics): formula $\Diamond\phi_1$ means that ϕ_1 holds sometime in the future, $\Box\phi_1$ says that ϕ_1 holds forever and lastly $\phi_1 \mathcal{W} \phi_2$ means that either ϕ_1 holds forever or sometime in the future ϕ_2 will hold and until that moment ϕ_1 holds (with ϕ_1 and ϕ_2 LTL formulas).

Templates *existence* and *absence* require that activity A occurs at least once and never occurs inside every process instance, respectively. Templates *choice* and *exclusive choice* indicate that A or B occur eventually in each process instance. The exclusive choice template is more restrictive because it forbids A and B to occur both in the same process instance. The *responded existence* template specifies that if A occurs, then B should also occur (either before or after A). The *response* template specifies that when A occurs, then B should eventually occur after A. The *precedence* template indicates that B should occur only if A has occurred before. Templates *alternate response* and *alternate precedence* strengthen the response and precedence templates respectively by specifying that activities must alternate without repetitions in between. Even stronger ordering relations are specified by templates *chain response* and *chain precedence*. These templates require that the occurrences of the two activities (A and B) are next to each other. Declare also includes some negative constraints to explicitly forbid the execution of activities. The *not responded existence* template indicates that if A occurs in a process instance, B cannot occur in the same instance. According to the *not response* template any occurrence of A cannot be eventually followed by B, whereas the *not precedence* template requires that any occurrence of B is not preceded by A. Finally, according to the *not chain response* and *not chain precedence*, A and B cannot occur one immediately after the other.

2.2 Analysis Tools for Declare

Several analysis plug-ins are available for Declare in the process mining tool ProM [24].[1] In [23, 26], the authors propose an approach for monitoring Declare models based on finite state automata. This approach provides the same functionalities described in this paper but it is limited to standard Declare specifications (i.e., data are not supported). Differently from their approach, in the technique presented here a compliance model can also include data-aware business constraints. In [35], the authors define *Timed Declare*, an extension of Declare based on a metric temporal logic semantics. The approach relies on timed automata to monitor metric dynamic constraints, but again data-aware specifications are not supported. As already mentioned in the introduction, in [29] the EC is used for defining a data-aware semantics for Declare. Moreover, in [30], the authors propose an approach for monitoring data-aware Declare constraints at runtime based on this semantics. This approach also allows the verification of metric temporal constraints, i.e., constraints specifying required delays and deadlines. This expressiveness comes with the limitation that the EC does not guarantee decidability when reasoning on the possible future outcomes of a partial trace, and hence is only used to check the actual events received so far. Automata-based techniques makes it instead possible to early-identify violations. In [25], the authors relies on a first-order variant of LTL to specify a limited version of data-aware patterns. Such extended patterns are used as the target language for a process discovery algorithm, which produces data-aware Declare constraints from raw event logs. Our FOLTL formalization for Declare extends the one presented in [25], and is tailored to a formal semantics that makes it suitable for the runtime monitoring of evolving traces.

[1] `www.processmining.org`

3. FO-LTL FOR DATA-AWARE DECLARE

Traditional LTL on finite traces is inadequate for expressing data-aware business constraints, as its propositional variables are not expressive enough to query complex data structures. We propose a first-order variant of LTL, called First-Order LTL (FOLTL), which merges the expressivity of first-order logic together with the LTL temporal modalities.

3.1 Syntax of FOLTL

We assume a relational representation of data and we define the *data schema* $\mathcal{S}$ as a set of relations, each one with an associated arity, and an interpretation domain Δ, which is a *fixed* a-priori and *finite* set of constants. A database instance I of $\mathcal{S}$ interprets each relation symbol with arity n as a subset of the cartesian product Δ^n. Values in Δ are interpreted as themselves, blurring the distinctions between constants and values. Given a schema $\mathcal{S}$, $\mathcal{I}$ denotes all possible database instances for $\mathcal{S}$.

DEFINITION 1 (FOLTL SYNTAX). *Given a data schema $\mathcal{S}$, the set of* closed FOLTL *formulas Φ obeys to the following syntax:*

$$\begin{array}{lcl}\Phi^\ell & := & true \mid Atom \mid \neg\Phi^\ell \mid \Phi^\ell_1 \wedge \Phi^\ell_2 \mid \forall x.\Phi^\ell \\ \Phi^t & := & \Phi^\ell \mid \bigcirc\Phi^t \mid \Phi^t_1 \,\mathcal{U}\, \Phi^t_2 \mid \neg\Phi^t \mid \Phi^t_1 \wedge \Phi^t_2 \\ \Phi & := & \Phi^t \mid \neg\Phi \mid \forall x.\Phi\end{array}$$

where x is a variable symbol and Atom is an atomic first-order formula or atom, i.e., a formula inductively defined as follows: (i) true is an atomic formula; (ii) if t_1 and t_2 are constants in Δ or variables, then $t_1 = t_2$ is an atomic formula, and (iii) if $t_1 \dots t_n$ are constants or variables and $R \in \mathcal{S}$ 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.

Intuitively, $\bigcirc\Phi$ (*next* Φ) says that Φ holds in the next instant, while $\Phi_1 \mathcal{U} \Phi_2$ (Φ_1 *until* Φ_2) says that there exists a future instant in which Φ_2 will hold and, until that moment, Φ_1 holds.

We define the logic 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 LTL temporal operators *finally* $\Diamond$, *globally* $\Box$ and *weak until* $\mathcal{W}$ are defined as: $\Diamond\Phi := true\,\mathcal{U}\,\Phi$; $\Box\Phi := \neg\Diamond\Phi$ and $\Phi_1 \mathcal{W} \Phi_2 := (\Phi_1 \mathcal{U} \Phi_2) \vee (\Box\Phi_1)$.

We observe that quantifiers for variables which occur in the scope of temporal operators are required to be in the front of the formula. We call such variables *across-state* variables.

3.2 Semantic of FOLTL over Finite Traces

Our analysis is not only based on finite traces, but it is performed at runtime, meaning while such traces are evolving. Roughly speaking, we assume a business process that produces events and possibly modifies data: each time it does so, we take the trace seen so far, i.e., the history of events and data, and we evaluate it considering that the process execution can still continue. This evolving aspect has a significant impact on the evaluation function: at each step, indeed, the monitor may return truth values which have a

degree of uncertainty due to the fact that future executions are yet unknown.

Without loss of generality, in what follows, we assume that the events generated by the process are stored in the database instance, and hence they are treated like data. A more detailed explanation on how we deal with events is given in Section 3.3. We now define the FOLTL semantics for finite traces that evaluates a FOLTL formula given a *finite* and *completed* trace. Then, we show how to use this semantics for building our evaluation function and monitoring *progressing* data instances.

Before showing the semantics of the language, we need to introduce the notion of assignment. 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 x. We denote with $\Phi[\eta]$ the formula obtained from Φ by replacing variables symbols with values in η.

A finite *trace* of length $n+1$ for a data schema $\mathcal{S}$ is a finite sequence $I_0, I_1, \ldots I_n$ of database instances over $\mathcal{S}$, i.e., is a function $\pi : \{0 \ldots n\} \to \mathcal{I}$ that assigns a database instance $\pi(i) \equiv I_i$ to each time instant $i \in \{0 \ldots n\}$.

DEFINITION 2 (FOLTL FINITE-TRACE SEMANTICS). *Given a* FOLTL *formula Φ over a schema $\mathcal{S}$, a domain Δ, an assignment η and a finite trace π of length $n+1$, we inductively define when Φ is true at an instant of time $0 \leq i \leq n$, written $(\pi, i, \eta) \models \Phi$, as follows:*

- $(\pi, i, \eta) \models true$;
- $(\pi, i, \eta) \models Atom$ *iff* $(\pi(i), \eta) \models Atom$, *where* $(\pi(i), \eta) \models Atom$ *is the usual* FO *evaluation function;*
- $(\pi, i, \eta) \models \neg\Phi$ *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* $d \in \Delta$, $(\pi, i, \eta_{x/d}) \models \Phi$;
- $(\pi, i, \eta) \models \bigcirc\Phi$ *iff* $i < n$ *and* $(\pi, i+1, \eta) \models \Phi$;
- $(\pi, i, \eta) \models \Phi_1 \mathcal{U} \Phi_2$ *iff for some* $i \leq j \leq n$ *we have* $(\pi, j, \eta) \models \Phi_2$ *and for all* $i \leq k < j$ *we have* $(\pi, k, \eta) \models \Phi_1$.

Furthermore, $(\pi, \eta) \models \Phi$ *iff* $(\pi, 0, \eta) \models \Phi$ *and, when Φ is closed (which is our assumption for constraints), we can simply write* $\pi \models \Phi$.

Notice that when a formula does not contain any temporal operator, its semantics corresponds to the traditional first-order semantics. Notice also that the domain Δ is the same for each instant of time (cf. [16] for a dissertation on different semantics for first-order modal logics). From the syntax, the semantics and the assumption of finite and fixed domain, we can translate every FOLTL formula into one in *temporal prenex normal form*, i.e., with all across-state quantifier in the front. From now on, we assume formulas to be in this form.

Since the interest of the verification community on runtime monitoring, different monitoring evaluation functions have been proposed [15, 10, 5]. Here, we adapt RV-LTL [5] to our first-order setting, and we call it RV-FOLTL. Such a semantics is tightly related to the notion of bad and good prefixes introduced in [22]. Given a FOLTL formula Φ, a *bad prefix* for Φ is a finite trace such that any (finite) extension of it does not satisfy Φ. In other words, no matter the continuation of the prefix, the formula Φ will always be evaluated to false since that moment on. Analogously, a *good prefix* can be defined as a finite trace which, no matter its continuation, will always satisfy (together with any continuation) property Φ.

Recalling that we are monitoring evolving executions, the definition of good and bad prefixes is particularly useful, because it allows us to evaluate the formula with a definitive truth value even if the trace is still evolving. Unfortunately, there are FOLTL formulas that do not have any good nor bad prefix, and hence cannot be evaluated as *true* or *false* until the current execution is actually finished. In such cases, we are still able to return a "temporary" truth value. In particular, we consider the partial trace π seen so far as if it was completed and we evaluate it according to the semantics in Definition 2: if π currently satisfies Φ but there is a possible prosecution of it that leads to falsifies Φ, then we say that Φ is *temporary satisfied*; if, instead, π currently falsifies Φ but there is a possible prosecution of it that leads to verify Φ, then we say that Φ is *temporary violated*.

Notice that, while evaluating a progressing trace, at each time instant we do not know whether the trace will still evolve, or the current instance seen is the last one, i.e., the process execution is finished. Without loss of generality, we introduce a propositional variable *Last* which is set to true by the external process when it terminates, and indicates that the current one is the closing instance of the trace.

DEFINITION 3 (RV-FOLTL SEMANTICS). *Given a* FOLTL *formula Φ and a finite trace π of current length n (but possibly still progressing), the monitoring evaluation function of a formula Φ on π, denoted by $[\pi \models \Phi]$, is an element of the set* $\{true, false, temp_true, temp_false\}$ *defined as follows:*

- $[\pi \models \Phi] := true$ *iff* $\pi(n) \models Last$ *and* $\pi \models \Phi$ *or* $\pi(last) \not\models Last$, $\pi \models \Phi$ *and for all finite possible prosecution σ we have that* $\pi\sigma \models \Phi$, *i.e., π is good prefix for Φ;*
- $[\pi \models \Phi] := false$ *iff* $\pi(n) \models Last$ *and* $\pi \not\models \Phi$ *or* $\pi(last) \not\models Last$, $\pi \not\models \Phi$ *and for all finite possible prosecution σ we have that* $\pi\sigma \not\models \Phi$, *i.e., π is bad prefix for Φ;*
- $[\pi \models \Phi] := temp_true$ *iff* $\pi(last) \not\models Last$, $\pi \models \Phi$ *and there exists a possible prosecution σ such that* $\pi\sigma \not\models \Phi$;
- $[\pi \models \Phi] := temp_false$ *iff* $\pi(last) \not\models Last$, $\pi \not\models \Phi$ *and there exists a possible prosecution σ such that* $\pi\sigma \models \Phi$.

No other cases are possible. In particular, if Φ is temporarily verified or temporarily falsified, there always exists both a prosecution that falsifies Φ and one that verifies it, otherwise π would be a good or bad prefix, respectively.

3.3 Declare Patterns in FOLTL

We now ground our FOLTL-based approach to the case of Declare, extending it to accommodate not only control-flow aspects, but also data-related ones, in the style of [28, 29, 30]. We remark that this is the first attempt to formalize data-aware Declare with temporal logics and exploit (finite state) automata-based techniques for monitoring.

As pointed out in [28, 29], a fundamental limitation of Declare in its basic form is the lack of data support: only (atomic) activities can be constrained. To overcome this limitation, [29] proposes an extension of the language to support data-aware activities and data conditions to augment the original Declare templates. The extension can be applied to both atomic and non-atomic activities, following the approach proposed in [30]. Here we focus on atomic activities only, briefly discussing the extension to non-atomic activities in Section 6.2. Nevertheless, the semantics we give here reconstructs faithfully the ideas presented in [30].

Table 2 shows how the basic Declare templates, extended with data, can be formalized using FO-LTL. The idea is to attach a payload that carries the data involved in execution of an activity. For example, the fact that customer *john* closes an order identified by 123 can be represented by the activity instance $close_order(john, 123)$. This corresponds to an instance of the activity $close_order/2$ with payload $\langle john, 123\rangle$, which in our model corresponds to a fact of relation $close_order(Cust, O_{id})$ (called *activity type*). We then assume that relations in $\mathcal{S}$ are partitioned into two sets: $\mathcal{A} \uplus \mathcal{R}$, where $\mathcal{A}$ is a set of activity relations (one per activity type), and $\mathcal{R}$ contains the other (business) relevant relations of the domain of interest. When monitoring a concrete system execution, the extension of such relations is manipulated as follows: every time an activity A is executed with payload $\vec{d} \in \Delta$, *(i)* the content of all relations in $\mathcal{A}$ is emptied, *(ii)* payload $\vec{d}$ is inserted into relation $A \in \mathcal{A}$, and *(iii)* the effects of the activity execution are incorporated, manipulating the extension of relations in $\mathcal{R}$ accordingly. Notice that this is a widely used mechanism to store payloads in data-aware processes (cf. [21]). To enforce the last bullet, we assume that the traced log of the system execution does not only lists which activities have been executed and with which payload, but also which facts are deleted and added by each activity execution. More specifically, we define an *event* as a tuple $\langle A(\vec{d}), ADD, DEL\rangle$, where A is an activity in $\mathcal{A}$, $\vec{d}$ is the payload of the executed activity instance, constituted by elements from Δ and such that its size is compatible with the arity of $A \in \mathcal{A}$, and ADD/DEL contain a set of facts over $\mathcal{R}$ that must respectively be added to and deleted from the current database. We assume that ADD has higher priority than DEL (i.e., if the same fact is asserted to be added and deleted in the same event, it is added).

With this notion at hand, we define a system log as a pair $\langle I_0, \mathcal{E}\rangle$, where I_0 is the (initial) database instance (defined in such a way that the extension of each activity type $A \in \mathcal{A}$ is empty), and $\mathcal{E}$ is a finite list of events $\langle e_1, \ldots, e_n\rangle$. A system log maps into a trace $I_0, I_1 \ldots I_n$ in the sense of Section 3.2 as follows: for each I_i with $i > 0$, given $e_i = \langle A(\vec{d}), ADD_i, DEL_i\rangle$, we have $I_i = (I_{i-1}|_{\mathcal{R}} \setminus DEL_i) \cup ADD_i \cup \{A(\vec{d})\}$, where $I_{i-1}|_{\mathcal{R}}$ is the database instance obtained from I_{i-1} by considering only tuples of relations $\mathcal{R}$. In this light, query $\xi_A(\vec{x}) = A(\vec{p}) \wedge \Phi(\vec{p}, \vec{y})$, issued over the current database I, returns *false* if A is not the last-executed activity, or the answer of $\Phi(\vec{d}, \vec{y})$ over I if A has been the last-executed activity, with payload $\vec{d}$. Queries of the form $\xi_A(\vec{x})$ are the basic building components for the FO-LTL-based formalization of data-aware Declare templates: they combine a test over the execution of the involved activity, together with a query over the current database. Such queries replace the activity name propositions used in standard Declare (cf. Table 1). As shown in Table 2, first-order quantification is used as follows.

Existence, absence and (exclusive) choice templates existentially quantify over $\vec{x}$, asserting that there must exist a state where the target activity (or one of the target activities) is executed so as to satisfy query Φ. For example, $existence(Close_order(c, o_{id}) \wedge Gold(c))$ models that at least one gold customer is expected to close an order during a system execution. All the other (binary) templates with source $\xi_A(\vec{x}) = A(\vec{p}_A) \wedge \Phi_A(\vec{x})$ (with $\vec{p}_A \subseteq \vec{x}$) and target $\xi_B(\vec{x}, \vec{y}) = A(\vec{p}_B) \wedge \Phi_B(\vec{x}, \vec{y})$ (with $\vec{p}_B \subseteq \vec{x}\vec{y}$) universally quantify over $\vec{x}$ (with scope the entire constraint), and existentially quantify over $\vec{y}$, asserting that for every payload $\vec{p}_A$ of A, and every query answer $\vec{x}$ to Φ_A, an execution of activity B is expected to happen by satisfying the dynamic constraint imposed by the template, as well as the involved data-aware conditions over $\vec{p}_B$ and $\Phi_B(\vec{x}, \vec{y})$. It is worth noting that both the payload of B as well as the query Φ_B could make use of some of the variables contained in $\vec{p}_A$ and/or $\vec{x}$. If this is the case, the common variables play the role of a *correlation* mechanism between the source and target activities/conditions. For example, constraint $response(Close_order(c, o_{id}), Pay_order(c', o_{id}) \wedge (c' = c \vee Responsible(c', c))$ states that whenever an order o_{id} is closed by customer c, *that* order must be eventually paid by either c herself, or by another customer c' who is responsible for c. Observe that the constraint universally quantify over c and o_{id}, whereas it existentially quantify over c'. Furthermore, o_{id} and c are used to correlate the source and target activities and conditions. Finally, as in the propositional case, negation templates express the negative version of relation templates.

Table 2: FO-LTL formalization for data-aware Declare; formula $\xi_A(\vec{x})$ corresponds to $A(\vec{p}) \wedge \Phi(\vec{x})$, where $\vec{p} \subseteq \vec{x}$, A is an activity type with payload $\vec{p}$, and $\Phi(\vec{x})$ is a first-order formula that queries the current database

PATTERN	FORMALIZATION
existence($\xi_A(\vec{x})$)	$\Diamond\exists\vec{x}.\xi_A(\vec{x})$
absence($\xi_A(\vec{x})$)	$\neg\Diamond\exists\vec{x}.\xi_A(\vec{x})$
choice($\xi_A(\vec{x})$,$\xi_B(\vec{y})$)	$\Diamond\exists\vec{x}.\xi_A(\vec{x}) \vee \Diamond\exists\vec{y}.\xi_B(\vec{y})$
exclusive choice($\xi_A(\vec{x})$,$\xi_B(\vec{y})$)	$\Diamond\exists\vec{x}.\xi_A(\vec{x}) \vee \Diamond\exists\vec{y}.\xi_B(\vec{y}) \wedge$ $\neg(\Diamond\exists\vec{x}.\xi_A(\vec{x}) \wedge \Diamond\exists\vec{y}.\xi_B(\vec{y}))$
responded existence($\xi_A(\vec{x})$,$\xi_B(\vec{x}, \vec{y})$)	$\forall\vec{x}.(\Diamond\xi_A(\vec{x}) \rightarrow \Diamond\exists\vec{y}.\xi_B(\vec{x}, \vec{y}))$
response($\xi_A(\vec{x})$,$\xi_B(\vec{x}, \vec{y})$)	$\forall\vec{x}.\Box(\xi_A(\vec{x}) \rightarrow \bigcirc\Diamond\exists\vec{y}.\xi_B(\vec{x}, \vec{y}))$
precedence($\xi_A(\vec{x}, \vec{y})$,$\xi_B(\vec{x})$)	$\forall\vec{x}.(\neg\xi_B(\vec{x})\,\mathcal{W}\,\exists\vec{y}.\xi_A(\vec{x}, \vec{y}))$
alternate response($\xi_A(\vec{x})$,$\xi_B(\vec{x}, \vec{y})$)	$\forall\vec{x}.\Box(\xi_A(\vec{x}) \rightarrow \bigcirc(\neg\xi_A(\vec{x})\,\mathcal{U}\,\exists\vec{y}.\xi_B(\vec{x}, \vec{y}))$
alternate precedence($\xi_A(\vec{x}, \vec{y})$,$\xi_B(\vec{x})$)	$\forall\vec{x}.(\neg\xi_B(\vec{x})\,\mathcal{W}\,\exists\vec{y}.\xi_A(\vec{x}, \vec{y}))\wedge$ $\forall\vec{x}.\Box(\xi_B(\vec{x}) \rightarrow \bigcirc(\neg\xi_B(\vec{x})\,\mathcal{W}\,\exists\vec{y}.\xi_A(\vec{x}, \vec{y}))$
chain response($\xi_A(\vec{x})$,$\xi_B(\vec{x}, \vec{y})$)	$\forall\vec{x}.\Box(\xi_A(\vec{x}) \rightarrow \bigcirc\exists\vec{y}.\xi_B(\vec{x}, \vec{y}))$
chain precedence($\xi_A(\vec{x}, \vec{y})$,$\xi_B(\vec{x})$)	$\forall\vec{x}.\Box(\bigcirc\xi_B(\vec{x}) \rightarrow \exists\vec{y}.\xi_A(\vec{x}, \vec{y}))$
not resp. existence($\xi_A(\vec{x})$,$\xi_B(\vec{x}, \vec{y})$)	$\forall\vec{x}.(\Diamond\xi_A(\vec{x}) \rightarrow \neg\Diamond\exists\vec{y}.\xi_B(\vec{x}, \vec{y}))$
not response($\xi_A(\vec{x})$,$\xi_B(\vec{x}, \vec{y})$)	$\forall\vec{x}.\Box(\xi_A(\vec{x}) \rightarrow \bigcirc\neg\Diamond\exists\vec{y}.\xi_B(\vec{x}, \vec{y}))$
not precedence($\xi_A(\vec{x}, \vec{y})$,$\xi_B(\vec{x})$)	$\forall\vec{x}.\Box(\exists\vec{y}.\xi_A(\vec{x}, \vec{y}) \rightarrow \neg\Diamond\xi_B(\vec{x}))$
not chain response($\xi_A(\vec{x})$,$\xi_B(\vec{x}, \vec{y})$)	$\forall\vec{x}.\Box(\xi_A(\vec{x}) \rightarrow \neg\bigcirc\exists\vec{y}.\xi_B(\vec{x}, \vec{y}))$
not chain precedence($\xi_A(\vec{x}, \vec{y})$,$\xi_B(\vec{x})$)	$\forall\vec{x}.\Box(\bigcirc\xi_B(\vec{x}) \rightarrow \neg\exists y.\xi_A(\vec{x}, \vec{y}))$

4. THE MONITORING APPROACH

In this section, we show how to actually build the monitor for data-aware Declare constraints. The problem of verifying (both offline and at runtime) data-aware temporal constraints is theoretically challenging, being undecidable in general. It requires knowledge of both verification and databases, and only recently has been actually addressed by the scientific community [20, 7, 9, 12]. Indeed, most of the literature on runtime monitoring focuses on checking propositional formulas [10, 5, 11], while the database community mainly studied offline analysis of temporal constraints on a database history [14, 8]. In [3], 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. Also, [19] investigates monitoring of first-order formulas, but a naive solution is adopted and no emphasis on complexity is placed.

4.1 Monitoring FOLTL Constraints

The recent work in [13] presents a flexible and automaton-based approach for runtime monitoring data-aware properties showing how to efficiently build a finite-state machine which recognizes good and bad prefixes of FOLTL formulas under the assumption of finite and fixed domain. One of the major issues when taking into account data is about efficiency. The main contribution of [13] is providing an EXPSPACE complexity in the size of the formula for building the monitor, while a naive approach would require exponential space in the size of the domain. Note that usually formulas are short, being written by humans, while the domain of constants (of a database) is large. We briefly discuss this approach and illustrate how to adapt it to our case.

The technique relies on the propositional Büchi automaton construction [2, 17] for a (propositional) LTL formula. However, since the language used in [13] is FOLTL, the notion of *first-order* (FO) automaton is introduced. A FO automaton is essentially built with the same procedure of a propositional Büchi automaton, but its transitions are labeled with first-order formulas with no temporal operators and its states contain data structures to smartly keep track of data. A Büchi automaton for a propositional LTL formula Φ_p is a finite state machine which accepts the infinite traces satisfying Φ_p by requiring that a final state is visited infinitely often. States of the automaton which does not lead to a cycle including a final state are marked as *bad*, as from them the accepting condition cannot ever be satisfied.

In order to recognize both the bad and good prefixes of a FOLTL formula Φ, two automata are needed: one is the FO automaton for Φ and the other one is the FO automaton for $\neg\Phi$. The first one is transformed into a final state machine which recognizes the bad prefixes of Φ by simply determining its bad states. Analogously, the automaton for $\neg\Phi$ is transformed into a finite state machine recognizing the bad prefixes of $\neg\Phi$, which are the good prefixes of Φ. The final monitor is the conjunction of the two.

As explained in [13], to monitor an evolving trace it is enough to navigate the resulting automaton's states: from the current state(s), transitions satisfying the current database instance and the data in the current state (recall that transitions are labeled with first-order formulas) are identified, and the new state(s) of the automaton are updated accordingly (more details are provided in Section 4.3). The semantics used in [13] is called LTL_3 [5], as it accounts three truth values: if a *bad* state is reached, the formula is falsified; if a *good* state is reached the formula is verified and if none of the two, the monitor returns the truth value ?. This semantics differs from the one given in Definition 3 but the technique in [13] is flexible enough to be extended to match our purposes. In what follows, we first illustrate the traditional technique to monitor propositional formulas with the RV-LTL semantics, and then we show how to adapt it to our first-order scenario using the ideas in [13].

4.2 Monitoring RV-LTL Constraints

Runtime verification of propositional LTL declarative process models under RV-LTL semantics has been studied in [27, 23]. The task is performed by means of a finite state machine which is built from a propositional LTL formula using the translation in [18]. This translation is, again, based on the propositional Büchi automaton for the formula and on the notion of good and bad prefixes. However, it presents significant differences with the analysis described in the previous section based on LTL_3 semantics. Indeed, while LTL_3 relies on the traditional infinite trace LTL semantics, the technique in [18] naturally follows from the finite LTL semantics in Definition 2. From the theoretical viewpoint, given a propositional LTL formula Φ_p, the finite-state machine for Φ_p obtained as described [18] recognizes all finite traces satisfying Φ_p. The algorithm simply navigates the automaton states and checks if the current state is final. If it is final, the trace is accepted, otherwise is rejected.

The work in [27, 23] adapts this algorithm to monitor evolving traces as follows. Automaton states are traversed as the trace is evolving, and at each step:

- if the execution is finished (i.e, *Last* is true) and the current state is final or the execution is not finished, the current state is final and there is no path from it reaching a non-final state, then return *true*;
- if the execution is finished (i.e, *Last* is true) and the current state is non-final or the execution is not finished, the current state is non-final and there is no path from it reaching a final state, then return $false$;
- if the trace is still evolving, the current state is final and there is a path from it reaching a non-final state, then return *temp_true*;
- if the trace is still evolving, the current state is non-final and there is a path from it reaching a final state, then return *temp_false*.

It is easy to see that this technique implements the semantics in Definition 3. The difference between LTL_3 and RV-LTL semantics is more evident when we consider what happens when *Last* holds. Since the semantics of LTL_3 is based on infinite traces, when the execution finishes and neither a good nor a bad prefix have been seen so far, the truth value of the monitoring remains undetermined, namely, ?. On the contrary the RV-LTL semantics guarantees that either *true* or $false$ is returned when the execution terminates.

4.3 FO Automaton for RV-LTL

The ideas presented in [13] are versatile and can be used on every kind of finite-state machine. We illustrate how they can be used to build an RV-LTL monitor for FOLTL formulas.

As a first step, since the domain of constants Δ is finite, a FOLTL formula Φ can be *propositionalized*, i.e., transformed into an *equivalent* propositional formula. Intuitively, the propositionalization (recursive) procedure p, when applied to a universal quantifier returns the conjunction of each assignments for the variables (analogously it returns the disjunction for the existential quantifier), and then it associates a propositional variable to each atom of Φ. Given that Φ can be translated in temporal prenex normal form, formula $\mathsf{p}(\Phi)$ has the structure $\bigwedge_{d_1\in\Delta}\bigvee_{d_2\in\Delta}\cdots\bigwedge_{d_{n+m}\in\Delta}\mathsf{p}(\Phi[x/d_1, y/d_2 \ldots z/d_{n+m}])$. This allows us to monitor $\Phi[x/d_1, y/d_2 \ldots z/d_{n+m}]$ for each assignment to across-state variables separately, which (when substituted to variables of Φ) returns a value among

$\wedge$	***true***	***false***
temp_true	*temp_true*	*false*
temp_false	*temp_true*	*false*

Table 3: Evaluation functions for the conjunction operator under the RV-LTL semantics

$\vee$	***true***	***false***
temp_true	*true*	*temp_true*
temp_false	*true*	*temp_false*

Table 4: Evaluation functions for the disjunction operator under the RV-LTL semantics

$\{true, false, temp_true, temp_false\}$. The result of the whole formula Φ is then obtained as in Table 3 and 4.

Based on the general technique presented in [13], we can use an automaton plus some auxiliary data structures to monitor each assignment. Let Φ be a FOLTL formula with all acoss-state quantifier in the front, being in temporal prenex normal form. We first get rid of them obtaining a formula $\hat{\Phi}$. Then, we build the monitor $\mathcal{A}$ for $\hat{\Phi}$ following the procedure in [18] by treating the atomic formulas of $\hat{\Phi}$ as propositional symbols. The resulting FO automaton $\mathcal{A}$ is likewise a propositional one, except for its transitions that are labeled with possibly *open* (because we got rid of the quantifiers) first-order formulas.

Also, we need to keep track of data. Let $\boldsymbol{\eta}$ be the set of all assignment to across-state variables for Φ. Each state s of $\mathcal{A}$ is marked with a set of assignments given by a marking function m, and at each step such a marking is updated according to the new database instance presented as input. At the beginning, the initial state s_0 is marked with all assignments, namely $m(s_0) = \boldsymbol{\eta}$. When a new event $e_i = \langle A(\vec{d_i}), ADD_i, DEL_i\rangle$ is presented as input, we first compute the new database instance I_i from e_i and I_{i-1} as described in Section 3.3. Then we check, for each state s of $\mathcal{A}$ and for each assignment $\eta \in m(s)$ which outgoing transition is satisfied by I_i. Recalling that a transition $s \to s'$ is labeled with a *open* first-order formula γ, we check whether $I_i \models \gamma[\eta]$: if it does, then we move the assignment η to state s'. When the new marking has been computed for all state, we perform the analysis described in the previous section, i.e., *for each assignment* η we check if there exists a path p that leads η to a final state, in order to assign a truth value in $\{true, false, temp_true, temp_false\}$ to η. Notice that in doing so, free variables of transitions involved in p has to be assigned according to η (cf. Section 5). Finally, we compose such values in order to evaluate the overall formula. This result is the output of the monitor and implements the RV-FOLTL runtime semantics in Definition 3.

5. APPLICATION TO WEB SECURITY

We believe the runtime monitoring technique presented here can be widely used in a broad range of security scenarios, where, to the best of our knowledge, no sophisticated analysis based on temporal formulas (and their corresponding automata) has been yet put in place.

For example, with the increasing attention of governments in open-intelligence analysis (OSINT), public forums administrators would decline their responsibility for risky behavior of group of users. To this purpose, administrators want to automatically check if (temporal) forum constraints are met by users. We assume a database schema containing the following set $\mathcal{R}$ of business relevant relations:

- $Users(usr, cntr)$, the list of people registered to the forum;
- $UntsdCntr(cntr)$, the list of countries whose government is not trusted by intergovernmental organizations;
- $BnndWrd(str)$, the list of banned words.

We also assume the system processes events of different types, which capture different activities performed by the users. Hence, we have the following activity relations $\mathcal{A}$:

- $\mathsf{Post}(usr, str)$ stores the payload of *post* events, which (singleton) tuple represents a user *usr* posting a new comment *str*;
- $\mathsf{Login}(usr, cntr)$ captures the *login* event of *usr* from a device located in *cntr*;
- $\mathsf{Delete}(usr, str)$ stores the payload of *delete* events, which deletes *str* previously written by *usr*.

Forum administrator checks the following constraints:

- alternate response: users cannot login outside their own country unless they delete all post showing banned words.
  $\forall usr, str.\Box((\mathsf{Post}(usr, str) \wedge BnndWrd(str)) \to$
  $\bigcirc(\neg(\exists cntr, usrc.\mathsf{Login}(usr, cntr) \wedge$
  $User(usr, usrc) \wedge cntr \neq usrc)\,\mathcal{U}\,\mathsf{Delete}(usr, str)))$
- Absence: user from untrusted countries cannot remove posts.
  $\neg\Diamond\exists usr, cntr, str.\mathsf{Delete}(usr, str) \wedge User(usr, cntr) \wedge UntsdCntr(cntr)$
- Response: posts with banned words have to be eventually deleted.
  $\forall usr, str.\Box((\mathsf{Post}(usr, str) \wedge BnndWrd(str) \to$
  $\Diamond\mathsf{Delete}(usr, str))$

Moreover we assume a sort of safety constraints that once a user is registered, it will be forever, and once an untrusted country is added, it will be forever, which can be expressed in FOLTL as: $\forall usr, cntr.\Box(User(usr, cntr) \to \Box User(usr, cntr))$ and $\forall cntr.\Box(UntsdCntr(cntr) \to \Box UntsdCntr(cntr))$ respectively. We observe that, even if the above formulas do not mention events (and indeed they do not follow any Declare pattern) they can still be expressed in FOLTL to constraint the evolution of data.

5.1 Construction of the FO Automaton

For the sake of simplicity, we now show how to get the FO automaton that monitors the conjunction of the absence and response formulas only, along with the two safety constraints above. The prenex normal form of their conjunction is:

$$\begin{aligned}\Phi:\ & \forall usr2, str2, usr3, cntr3, cntr4.\\ & (\Box(\neg\exists usr, cntr, str.(Delete(usr, str) \wedge\\ & User(usr, cntr) \wedge UntsdCntr(cntr))) \wedge\\ & \Box((Post(usr2, str2) \wedge BnndWrd(str2) \to\\ & \Diamond Delete(usr2, str2))) \wedge\\ & \Box(User(usr3, cntr3) \to \Box User(usr3, cntr3)) \wedge\\ & \Box(UntsdCntr(cntr4) \to \Box UntsdCntr(cntr4)))\end{aligned}$$

Then, we discard all the quantifiers in the front and we build the FO automaton of the formula considering atoms as propositional variables. The FO automaton for Φ has 8 states and 72 transitions. For lack of space Figure 1 shows only the fragment of the automaton that is meaningful for our example (we pruned transitions which do not satisfy the antecedent of the safety constraints and transitions leading

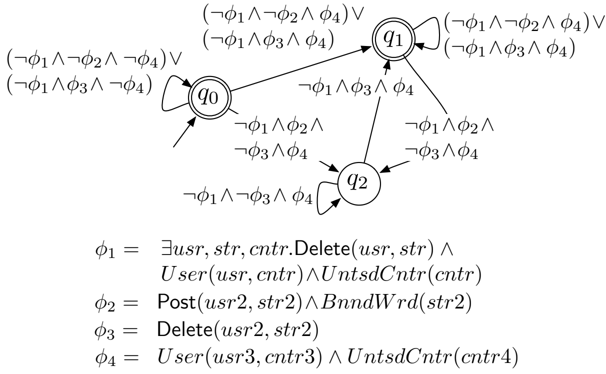

Figure 1: Partial FO automaton for formula Φ in Section 5

to non-final sink states). Double circled states q_0 and q_1 are final, and q_0 is also initial.

5.2 Monitoring Sample Process Instances

In what follows, we consider domain $\Delta = \{u1, u2, c1, c2, str1, str2\}$, the following initial database instance I_0:

$$\left\{ \begin{array}{lll} User(u1, c1) & User(u2, c2) & UntstdCntr(c2) \\ BnndWrd(s1) & & \end{array} \right\}$$

and we simulate the execution of the algorithm as new events are presented as input.

Let $\boldsymbol{\eta}$ be the set of all assignment for the across-states variables ($usr2$, $str2$, $usr3$, $cntr3$, and $cntr4$) of Φ with values in Δ. At the beginning, the marking function m is: $m(q_0) = \boldsymbol{\eta}$, $m(q_1) = \emptyset$ and $m(q_2) = \emptyset$. When event $e_1 = \langle \mathsf{Post}(u1, s5), \emptyset, \emptyset \rangle$ is processed, the resulting database instance I_1 is:

$$\left\{ \begin{array}{lll} User(u1, c1) & User(u2, c2) & UntstdCntr(c2) \\ BnndWrd(s1) & \mathsf{Post}(u1, s5) & \end{array} \right\}$$

and the new marking is $m(q_0) = \boldsymbol{\eta}_1$, $m(q_1) = \boldsymbol{\eta}_2$ and $m(q_2) = \emptyset$, where $\boldsymbol{\eta}_1 = \{\eta \mid (I_1, \eta) \models (\neg\phi_1 \wedge \neg\phi_2 \wedge \neg\phi_4) \vee (\neg\phi_1 \wedge \phi_3 \wedge \neg\phi_4)\}$, i.e., the set of assignments satisfying the formula in looping transition in q_0 given I_1 (e.g., assignment $\langle usr2/s2, str2/s2, usr3/s2, cntr3/s2, cntr4/s2 \rangle$) and $\boldsymbol{\eta}_2 = \{\eta \mid (I_1, \eta) \models (\neg\phi_1 \wedge \neg\phi_2 \wedge \phi_4) \vee (\neg\phi_1 \wedge \phi_3 \wedge \phi_4)\}$, i.e., the set of assignments satisfying the formula on transition from q_0 to q_1 given I_1 (e.g., assignment $\langle usr2/u2, str2/s2, usr3/u2, cntr3/c2, cntr4/c2 \rangle$). Since both q_0 and q_1 are final states, no further analysis is required to check if a final state is reachable and the output of the monitor is *temp_true*. In this case, the fixed and finite domain assumption is not too strict: indeed, even though $s5 \notin \Delta$, the algorithm still works correctly, since strings which "activate" the constraints are those matching banned words (that, by consequence are already in the domain).

The next event is $e_2 = \langle \mathsf{Post}(u2, s2), \{BnndWrd(s2)\}, \emptyset \rangle$, hence, we get the following database instance I_2:

$$\left\{ \begin{array}{lll} User(u1, c1) & User(u2, c2) & UntstdCntr(c2) \\ BnndWrd(s1) & BnndWrd(s2) & \mathsf{Post}(u2, s2) \end{array} \right\}$$

Notice that not only a new event is seen, but also data has changed (a new banned word has been added). Now, a user of an untrusted country is posting something bad. All assignment in q_1 with $usr2/u2$ and $str2/s2$, by satisfying transition $\neg\phi_1 \wedge \phi_2 \wedge \neg\phi_3 \wedge \phi_4$, move to state q_2. Let us focus on one of such assignments, namely $\bar{\eta} = \langle usr2/u2, str2/s2, usr3/u2, cntr3/c2, cntr4/c2 \rangle$. We have to compute the truth value for $\bar{\eta}$, which means we have to check if there is a path from q_2 that can reach a final state (q_1 in this case, as q_0 cannot be reached from q_2). Assignment $\bar{\eta}$ will not ever satisfy transition $\neg\phi_1 \wedge \phi_3 \wedge \phi_4$ (that is the only path to q_1) as it contains the contradiction $\neg\exists usr, str, cntr.\mathsf{Delete}(usr, str \wedge User(usr, cntr) \wedge UntsdCntr(cntr)) \wedge \mathsf{Delete}(u2, s2) \wedge User(u2, c2) \wedge UntsdCntr(c2)$ when its free variables are substituted according to $\bar{\eta}$. From the semantics in Definition 3, $\bar{\eta}$ is evaluated to $false$. The truth value of Φ is the conjunction of the truth value of each assignment, being all its variables universally quantified, hence resulting in $false$ as shown by Table 3.

This result may come unexpected, since there is no constraint forcing users from untrusted countries to post no banned words. However, this is an example of *early detection*: nothing bad has happened yet, but the reasoning capabilities enabled by the use of temporal logics and automata realize that there exists no prosecution of the current trace that satisfies the constraints. Indeed, our temporal analysis recognizes that the conjunction of the absence constraint (users from untrusted countries cannot delete posts) with the response (post with banned words have to be eventually deleted) and the two safety constraints (users and banned words cannot be deleted from the database), entails that as soon as a user from an untrusted country posts a banned word, the constraints are violated.

Few other remarks are in order. First of all, when the monitor returns $true$ or $false$, the runtime analysis can be stopped, as from Definition 3, such truth values will not change regardless of future prosecution of the trace. Also, notice that this approach does not require the whole trace to evaluate a formula, but the current database instance only: data are kept in the automaton as assignments to the formula to be monitored, and in such a way that only data relevant to evaluate this formula are stored. Finally, in many scenarios such as the one described in this example, the finite and fixed domain assumption is not too strict. On the one hand, usually the active domain of the initial database instance I_0 is taken as the monitoring domain, and it can also be augmented with other business relevant constants. On the other hand, it is the cost to be paid in order to gain in decidability and in efficiency: when we relax this assumption, the monitoring task becomes undecidable, as shown in [4].

6. DISCUSSION

We conclude the analysis of our framework by discussing its applicability to monitoring concrete system traces stored in the XES format [1], and its versatility in modeling dynamic constraints that go beyond data-aware Declare.

6.1 Monitoring XES Traces

Recently, the IEEE Task Force on Process Mining has proposed XES [1] as a standard, XML-based format for representing (process) execution traces. It is the result of a thorough analysis of concrete process-aware information systems and the kind of information they log.

A XES log is minimally conceived as consisting of a set of traces (i.e., specific process instances), which, in turn,

are described by sequences of events. Each of these three concepts can be further described by an arbitrary set of attributes that maintain the actual data. To attach specific semantics to data in an event log, XES introduces the concept of extension. An extension defines a number of standardized attributes for each level in the hierarchy (e.g., log, trace, attributes), together with their type (e.g., string, boolean) and their specific attribute keys. For example, the XML snippet

```
<event>
  <string key="concept:name" value="pay_order"/>
  <date key="time:timestamp" value="..."/>
  <string key="org:resource" value="john"/>
  <string key="Order ID" value="123"/>
</event>
```

models an event attesting that John has paid the order identified by code 123. To do so, it exploits three of the predefined XES extensions: 1. *concept extension*, so as to include the name of the executed activity; 2. *time extension*, so as to store the time point in which the event occurred; 3. *organizational extension*, so as to qualify the name/role/group of the resource that has triggered the event. The main difference between this kind of event and the notion of event used in Section 3.3 is that in the XES snippet there is no information about the facts that are deleted and added by the event. This can however be seamlessly captured in XES by defining a specific *DB manipulation extension* that adds two children elements to `event`: a complex attribute `toDel` to manage tuples that must be deleted, and a complex attribute `toAdd` to manage tuples that must be added. Such elements, in turn, contain a set of `tuple` elements, each denoting a tuple, constituted by a relation name and a list of (named) fields (or, in the case of deletion, simply by the primary key of the tuple). For example,

```
<event>
  <string key="concept:name" value="pay_order"/>
  <date key="time:timestamp" value="..."/>
  <string key="org:resource" value="john"/>
  <string key="Order ID" value="123"/>
  <string key="toAdd">
      <tuple relation="CLOSED-ORDERS">
          <field key="OID" value="123"/>
          <field key="OWNER" value="john"/>
      </tuple>
  </string>
  <string key="toDel">
      <tuple relation="PENDING-ORDERS" PK="123"/>
  </string>
</event>
```

extends the aforementioned XES event by stating that the event has the effect of moving order 123 from table `PENDING-ORDERS` to table `CLOSED-ORDERS`, also inserting the information about the owner (John).

6.2 Modeling Extended Constraints

We observe that our approach is versatile enough to seamlessly capture dynamic constraints that go beyond those addressed by Declare. First, notice that the distinction between activities and other relations is, in our framework, only done for modeling convenience, but both aspects are treated uniformly as relations in the data schema. This makes directly possible to model dynamic constraints that single out the expected/forbidden evolutions of data, independently from the actual executed activities. E.g., constraint $\Box\forall c.(\neg Gold(c)\,\mathcal{W}\,\exists c_1, c_2.c \neq c_1 \wedge c \neq c_2 \wedge c_1 \neq c_2 \wedge Responsible(c, c_1) \wedge Responsible(c, c_2))$ is a data-driven *precedence* stating that customers can become gold only if they become responsible of at least two other customers.

A second important extension concerns the possibility of tackling cross-instance constraints, i.e., dynamic constraints spanning multiple process instances. From the logging point of view, this requires to see the entire system log as a unique execution trace, and at the same time to attach an explicit instance identifier to each logged events, so as to enable correlation among events of the same process instance. There are various techniques to track this kind of information, such as WS-Addressing in the context of web service interaction. From the language point of view, it is possible to model constraints that are applied on a per-instance basis (by just correlating on the process instance identifier), or constraints that possibly cross multiple instances. E.g., $response(Close_order(pid, c, o_{ID}), Pay_order(pid, c', o_{ID}) \wedge (c' = c \vee Responsible(c', c))$ reconstructs the *response* constraint modeled in Section 3.3 by correlating on the instance identifier *pid*, whereas $not\ response(Block(pid_1, c, a) \wedge Admin(a), Open_order(pid_2, o, c))$ models a cross-instance *negation response* stating that whenever a customer is blocked by a certain administrator, she cannot open orders anymore (notice the correlation on the customer variable c, but not on the process identifiers, which can be possibly different). A similar approach can be used to model activity-based correlation, and in turn support non-atomic activities in the style of [30].

7. CONCLUSIONS

The framework presented in this paper represents the first attempt of exploiting an automata-based approach for the runtime monitoring of process execution traces against dynamic, first-order constraints, which are able to accommodate a plethora of different templates, including all those of the Declare language, extended with data-related aspects. The framework currently assumes a finite, rigid quantification domain for constraints, which is parsimoniously handled by the monitoring technique. We are currently studying an extension for dealing with varying domains. At the same time, an implementation of our technique is currently under testing. It has been developed as an extension of the automata-based approach for standard Declare, in the form of an operational support provider inside the well-known ProM process mining framework.

For the time being, the monitoring technique supports all the fine-grained truth values of RV-LTL. The next step is to extend it to provide *continuous support* (to provide verification capabilities even after a violation has taken place) and *advanced diagnostics*, starting from the LTL-based approach in [23] and lifting it to the case of FOLTL.

8. REFERENCES

[1] XES Standard Definition, 2009. www.xes-standard.org.

[2] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[3] D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In *CAV*, pages 1–18, 2010.

[4] A. Bauer, J.-C. Küster, and G. Vegliach. From propositional to first-order monitoring. In *RV*, pages 59–75, 2013.

[5] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *RV*, pages 126–138, 2007.

[6] 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.

[7] F. Belardinelli, A. Lomuscio, and F. Patrizi. Verification of deployed artifact systems via data abstraction. In *ICSOC*, pages 142–156, 2011.

[8] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems*, 20(2):149–186, 1995.

[9] E. Damaggio, A. Deutsch, R. Hull, and V. Vianu. Automatic verification of data-centric business processes. In *BPM*, pages 3–16, 2011.

[10] M. d'Amorim and G. Rosu. Efficient monitoring of omega-languages. In *CAV*, pages 364–378, 2005.

[11] 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.

[12] 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.

[13] R. De Masellis and J. Su. Runtime enforcement of first-order ltl properties on data-aware business processes. In *ICSOC*, pages 54–68, 2013.

[14] G. Dong, J. Su, and R. Topor. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14:187–223, 1995.

[15] 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.

[16] M. Fitting and R. L. Mendelsohn. *First-Order Modal Logic*. Kluwer Academic Press, 1998.

[17] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV*, pages 3–18, 1995.

[18] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *ASE*, pages 412–416, 2001.

[19] S. Hallé and R. Villemaire. Runtime monitoring of message-based workflows with data. In *EDOC*, pages 63–72, 2008.

[20] B. B. Hariri, D. Calvanese, G. De Giacomo, R. De Masellis, and P. Felli. Foundations of relational artifacts verification. In *BPM*, pages 379–395, 2011.

[21] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. H. Linehan, S. Maradugu, A. Nigam, P. N. Sukaviriya, and R. Vaculín. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In *DEBS*, pages 51–62, 2011.

[22] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[23] F. Maggi, M. Montali, M. Westergaard, and W. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *BPM 2011*, volume 6896, pages 132–147, 2011.

[24] F. M. Maggi. Declarative process mining with the declare component of prom. In *BPM (Demos)*, 2013.

[25] F. M. Maggi, M. Dumas, L. García-Bañuelos, and M. Montali. Discovering data-aware declarative process models from event logs. In *BPM*, volume 8094 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2013.

[26] F. M. Maggi, M. Westergaard, M. Montali, and W. M. P. van der Aalst. Runtime verification of LTL-based declarative process models. In *RV 2011*, volume 7186, pages 131–146.

[27] F. M. Maggi, M. Westergaard, M. Montali, and W. M. P. van der Aalst. Runtime verification of ltl-based declarative process models. In *RV*, pages 131–146, 2011.

[28] M. Montali. *Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach*, volume 56 of *Lecture Notes in Business Information Processing*. Springer, 2010.

[29] M. Montali, F. Chesani, F. M. Maggi, and P. Mello. Towards data-aware constraints in declare. In *SAC*, pages 1391–1396. ACM Press and Addison Wesley, 2013.

[30] M. Montali, F. M. Maggi, F. Chesani, P. Mello, and W. M. P. van der Aalst. Monitoring business constraints with the event calculus. *ACM TIST*, 5(1):17, 2013.

[31] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative Specification and Verification of Service Choreographies. *ACM Transactions on the Web*, 4(1), 2010.

[32] M. Pesic, H. Schonenberg, and W. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *Proc. of EDOC*, pages 287–300. IEEE, 2007.

[33] W. van der Aalst, M. Pesic, and H. Schonenberg. Declarative Workflows: Balancing Between Flexibility and Support. *Computer Science - R&D*, pages 99–113, 2009.

[34] M. Westergaard and F. M. Maggi. Declare: A tool suite for declarative workflow modeling and enactment. In *BPM (Demos)*, 2011.

[35] M. Westergaard and F. M. Maggi. Looking into the future: Using timed automata to provide a priori advice about timed declarative process models. In *OTM*, volume 7565 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2012.