Runtime Monitoring of Message-Based Workflows with Data

Sylvain Hallé and Roger Villemaire
Université du Québec à Montréal
C. P. 8888, Succ. Centre-Ville
Montréal, Canada H3C 3P8
halle@info.uqam.ca *

Abstract

We present an algorithm for the runtime monitoring of business process properties with data parameterization. The properties are expressed in LTL-FO$^+$, an extension to traditional Linear Temporal Logic that includes full first-order quantification over the data inside a trace of XML messages. The algorithm works "on-the-fly": it keeps in memory only the states that are necessary at each step. Initial results indicate that LTL-FO$^+$ is an appropriate language for expressing data dependencies on message traces and that its processing overhead on sample traces is acceptable.

1 Introduction

Message-based workflows are structures where the input and output of operations is composed of "messages" formed of data elements. Document-passing business processes, method calls in programming languages and XML-based web service interactions can all be modelled by this general representation. In [16], it was shown how such message-based workflows are subject to "data-aware" constraints where the sequence of messages and their content are interdependent. To this end, an extension over classical Computation Tree Logic called CTL-FO$^+$ was introduced. CTL-FO$^+$ includes first-order quantification on message content in addition to temporal operators; it was shown to be adequate for expressing data-aware constraints.

Although appropriate algorithms for the static validation of data-aware constraints have been presented and tested [17], there remains situations in which such validation is not possible or desirable for various reasons. Therefore, alternative solutions for ensuring the correctness of message-based workflows must be developed. One of them is to engage in *runtime monitoring*, which consists of the observation of a workflow's execution in real time, accompanied with a verification that it conforms to some specification.

In this paper, we present an algorithm for the runtime monitoring of data-aware workflow constraints. In Section 2, we advocate for the use of runtime monitoring techniques and present a number of real-world scenarios taken from the literature in which the specifications are data-aware temporal properties. To express these properties using a uniform formal notation, in Section 3 we introduce LTL-FO$^+$, a counterpart to CTL-FO$^+$ for expressing data-aware properties of execution *traces*. LTL-FO$^+$ is a linear temporal logic augmented with full first-order quantification over the data inside a trace of XML messages. We show how LTL-FO$^+$ is suitable for expressing the constraints previously described and explain how it differs from related works on the subject.

Section 4 is the core of this paper: it describes the runtime monitoring algorithm for LTL-FO$^+$. This algorithm distinguishes itself from existing approaches in two respects: 1) the algorithm allows the monitored properties to quantify over data fields inside messages; 2) it works "on-the-fly", without the need to pre-compute an automaton, to store previous messages or to keep in memory anything except its current state.

To assess the feasibility of LTL-FO$^+$ runtime monitoring in practical contexts, we performed a set of experiments on some of the properties mentioned in Section 2. In Section 5, the results of these experiments are presented and discussed. Initial findings indicate that runtime monitoring of LTL-FO$^+$ can be performed in real-world scenarios, and that augmenting existing specification languages with quantification over data does not impose a large overhead on their processing, even for transactions of 1,000 messages and domains of over 500 elements. For all these reasons, the runtime monitoring of data-aware properties on a message-based workflow can be seen as a viable complementary approach to its static validation.

*We gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada on this research.

2 Motivation

The runtime monitoring of a message-based workflow is a process which presents itself in various ways. The observation of the execution can be made by an external agent intercepting messages to and from a given party, or be conducted by calling internal monitoring code. Similarly, the observed trace of messages can be analyzed once the execution is completed or can be performed step-by-step in parallel with the execution of the actual workflow. Despite these differences, a number of observations apply to all types of runtime monitoring frameworks.

2.1 A Case for Workflow Runtime Monitoring

Several arguments in favour of runtime monitoring approaches have been put forward [14]. First, the satisfaction of requirements sometimes depends on assumptions on the partners that cannot be verified prior to the actual implementation of the system. In the particular case of service-oriented architectures, partners can be discovered dynamically and can even change drastically during execution, invalidating the assumptions on which the original process was deemed correct.

Moreover, in some occasions, a static, *a priori* model checking of the intended process is simply impossible or intractable because of the size of data domains [20]. For example, on the theoretical level, web services communicate through channels of potentially infinite length, thereby rendering the general model checking problem undecidable without resorting to some form of simplification or abstraction of the original model. A runtime monitor has the advantage of working with the actual implementation of a process.

Finally, even cases where model checking is possible can present a challenge. The partners involved in a business process can be implemented in heterogeneous languages and formalisms that make it hard to have a uniform, global picture of the whole conversation suitable for a static verification; [10] describes a system combining BPEL processes with Java-based partners and concludes that static analysis approaches do not handle such features well. There also exist situations at runtime which, although they do not constitute strict violations of a specification, must be addressed as soon as they are discovered: [2] gives the example of an online shop being refused a money transfer by its partner bank, or of a client repeatedly asking for products that are no longer in stock.

Independent of these technical aspects, the runtime monitoring of a process is also sound business-wise. [23] remarks that monitoring can increase trust in an electronic marketplace by providing the consumer of a service the ability to check by itself the transaction that takes place.

2.2 Sample Runtime Monitoring Scenarios

To motivate the need for a data-aware runtime monitoring algorithm, we present below three simple scenarios taken from existing literature, and in which runtime monitoring of properties is needed. In each of these scenarios, the authors enumerated a list of constraints on the pattern of exchanged messages that have to be monitored. The constraints in each scenario were actually expressed in a different language by their respective authors; they are paraphrased in English for the sake of readability.

2.2.1 Holiday Location-Finder

In [14], a simple web service for finding hotels is described. A user sends a request for a particular point in geographical coordinates, provides the maximum distance around that point where desired hotels should be, and gets in return a list of locations with their coordinates. This interaction is modelled as sequence of messages with data content. The input message is of the form:

<RequestSpec>
  <departureLocation> L </departureLocation>
  <maxDistance> 123 </maxDistance>
  <criteria> $\cdots$ </criteria>
</RequestSpec>

And the returned message is the following:

<LocationResults>
  <location> L_1 </location>
  ...
  <location> L_n </location>
</LocationResults>

The paper cites a context in which a customer invokes the service under a temporary trial license, and wishes to monitor the interaction to verify that the location finder delivers what it promises. A property to monitor in this situation is the following:

Runtime Property 1 (From [14])**.** *For all the returned locations, the distance from the requested point should be no more than the requested distance.*

This first example constitutes the most direct application of message-based workflows: the modelling of XML request-response patterns in web service transactions.

2.2.2 User-Controlled Lightpaths

The User-Controlled Lightpath (UCLP) research project [5] allows end users to self provision and dynamically reconfigure optical networks resources; these resources are virtualized and exposed to the end user as instances of web

services that implement functionalities related to lightpath manipulation.

Lightpaths can be partitioned and concatenated by means of web service invocations. Typical messages for concatenation and partition request are structured as follows:

<message>
<operation>concatenateRequest</operation>
<LPO-ID>i_1</LPO-ID>
<LPO-ID>i_2</LPO-ID>
...
<LPO-ID>i_n</LPO-ID>
</message>

<message>
<operation>partitionRequest</operation>
<LPO-ID>i</LPO-ID>
<bandwidth>b</bandwidth>
</message>

In [16], it was shown how specific constraints must be respected on the invocation of these methods. For example, the original LPO to be partitioned "ceases to exist" as long as it is not un-partitioned; it cannot appear in further operations. Therefore, a sample constraint from this environment is the following:

Runtime Property 2 (From [16])**.** *Any LPO ID appearing in any partition request must be different from any LPO ID appearing in any future concatenate request.*

This second example is still web service-based; however, the reader should be aware that the web service paradigm is simply used as a wrapper around management functionalities for physical resources.

2.2.3 Car Rental System

As a last example of runtime properties, we cite the example of a car rental system presented by [20]. In this context, users can request a given car, hire a car, and enter and exit a parking lot with their car. These actions are represented as predicates in the Event Calculus; however, for the needs of the presentation, such predicates can be encoded as XML documents. Hence, the fact that cars $c_1, \ldots, c_n$ enter parking p can be expressed as the following "message":

<message>
<action>enter</action>
<car>c_1</car>
...
<car>c_n</car>
<parking>p</parking>
</message>

A property of the CRS expresses an assumption about the behaviour of the parking lot sensors that needs to be monitored for problems:

Runtime Property 3 (From [20])**.** *Every car c_i entering a parking lot p must depart p before entering any other parking lot.*

This last example shows how specific features of systems modelled with other formalisms can be translated into message-based workflows. Moreover, it shows a property in which the time span between the two messages that are considered (enter and exit) is arbitrary and unknown in advance.

In each of these scenarios, additional constraints can complexify the monitoring process: asynchronous communications, lost, delayed or out-of-order messages can distort an otherwise valid interaction, without it being any of the services' "fault". In this paper, a focus has been placed on providing means of *detecting* that an assumption on the communication has been violated for one of the collaborating services, and not on repairing an invalid transaction or determining the actual cause of the violation.

3 A Logic for Message Traces with Data Parameters

Runtime monitoring properties are generally expressed with a variant of a state machine or temporal logic. However, few of these representations allow a full quantification over the data fields of the messages, although there exists situations in which such quantification is necessary. To address our point, we highlight the common points of the previous three scenarios: 1) They involve properties referencing data elements inside exchanged "messages"; 2) These data elements are taken at two different moments in the execution and need to be compared; 3) The data elements cannot be enumerated statically; therefore, a form of quantification over data must be possible.

In [16], the logic CTL-FO$^+$ was introduced as an appropriate formalism for expressing these "data-aware" workflow properties. However, CTL-FO$^+$ is aimed at model checking of workflow properties; all CTL-FO$^+$ sentences are of the form "for every execution path starting from the current state", or "there exists an execution path starting from the current state". In a runtime monitoring context, since we are observing a single trace at a time, such quantifiers are not necessary. The logic we present in this section is LTL-FO$^+$, a counterpart to CTL-FO$^+$ to express properties on single execution paths.

3.1 Messages and Traces

The logic will be defined in relation to a suitable model of a transaction. In the present context, this suitable representation should model the actual messages that are exchanged.

We start by defining a set Π of parameters and a set Ω of values that are used to represent the content of the messages. We define a special symbol # that stands for "no-value". Couples of parameters and values form a message element:

Definition 1 (Message elements)**.** *The set of defined message elements is $\mathcal{E}_d = \Pi \times (\Omega \cup \{\#\})$. We also consider the set of undefined message elements, which is the singleton $\mathcal{E}_u = \{(\#, \#)\}$. A* message element *is a member of $\mathcal{E} = \mathcal{E}_d \cup \mathcal{E}_u$.*

The concept of message element closely parallels the structure of a (flat) XML message. The parameters stand for the tag names, while the values represent the data inside the tag. A message is then an ordered sequence of message elements:

Definition 2 (k-messages)**.** *Let k be a positive integer, and for $i < k$, define $D_i = \{(e_1, \ldots, e_k) : e_i \in \mathcal{E}_u \wedge e_{i+1} \in \mathcal{E}_d\}$. The set of k-messages is defined as $M_k = \mathcal{E}^k \setminus (\bigcup_{i=1}^{k-1} D_i)$.*

Note that this representation does not allow nested tags. We shall see later on how nested structures can be taken into account. It suffices then to formally define a trace in this context:

Definition 3 (Message trace)**.** *A* message trace *(also simply called a* trace*) is a sequence $\rho = m_1 m_2 \ldots$ such that $m_i \in M_k$ for $i \geq 1$.*

We define ρ_i to be the i-th message of the trace ρ, and ρ^i the trace obtained from ρ by starting at the i-th message.

Definition 4 (Domain expression)**.** *Let $m = ((p_1, v_1), \ldots, (p_k, v_k)) \in M_k$ be a k-message and $p \in \Pi$ be a parameter name. The function $Dom : M_k \times \Pi \to 2^\Omega$ is defined as follows: $Dom_m(p) = \{v_i : p_i = p\}$.*

The function Dom simply fetches all values of a given tag in a message. For example, in the following message m, we have $Dom_m(item) = \{A, B, C\}$.

```
<message>
  <item>A</item>
  <item>B</item>
  <item>C</item>
  <client>10</client>
</message>
```

Since the number of message elements is bounded by k, the domain returned by Dom contains at most k elements.

3.2 Syntax and Semantics of LTL-FO$^+$

Using this model of exchanged messages, we can now define the syntax and semantics of the Linear Temporal Logic with Full First-order Quantification (LTL-FO$^+$), an extension of the well-known temporal logic LTL [7]. Many major model checking tools such as SPIN [18] verify temporal formulæ expressed in LTL. The reader is referred to [7] for a deeper coverage of LTL and other temporal logics.

Definition 5 (Syntax)**.** *The language LTL-FO$^+$ (Linear Temporal Logic with Full First-order Quantification) is obtained by closing LTL under the following construction rules:*

1. *If x and y are variables or constants, then $x = y$ is a LTL-FO$^+$ formula;*
2. *If φ and ψ are LTL-FO$^+$ formulæ, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \to \psi$, $\mathbf{G}\,\varphi$, $\mathbf{F}\,\varphi$, $\mathbf{X}\,\varphi$, $\varphi\,\mathbf{U}\,\psi$, $\varphi\,\mathbf{V}\,\psi$ are LTL-FO$^+$ formulæ;*
3. *If φ is a LTL-FO$^+$ formula, x_i is a free variable in φ, $p \in \Pi$ is a parameter name, then $\exists_p x_i : \varphi$ and $\forall_p x_i : \varphi$ are LTL-FO$^+$ formulæ.*

Definition 6 (Semantics)**.** *We say a message trace ρ* satisfies *the LTL-FO$^+$ formula φ, and write $\rho \models \varphi$ if and only if it respects the following rules:*

$$
\begin{aligned}
\rho \models c_1 = c_2 &\quad\Leftrightarrow\quad c_1 \text{ is equal to } c_2 \\
\rho \models \neg\varphi &\quad\Leftrightarrow\quad \rho \not\models \varphi \\
\rho \models \varphi \vee \psi &\quad\Leftrightarrow\quad \rho \models \varphi \text{ or } \rho \models \psi \\
\rho \models \mathbf{F}\,\varphi &\quad\Leftrightarrow\quad \rho^i \models \varphi \text{ for some } i \leq 1 \\
\rho \models \mathbf{X}\,\varphi &\quad\Leftrightarrow\quad \rho^2 \models \varphi \\
\rho \models \varphi\,\mathbf{U}\,\psi &\quad\Leftrightarrow\quad \rho_j \models \psi \text{ for some } j \text{ and } \rho_i \models \varphi \\
&\qquad\qquad \text{for } i < j \\
\rho \models \exists_p x_i : \varphi &\quad\Leftrightarrow\quad \rho \models \varphi[b/x_i] \text{ for some } b \in Dom_{\rho_1}(p)
\end{aligned}
$$

As usual, we define the semantics of the other connectors with the following identities:

$$
\begin{aligned}
\varphi \wedge \psi &\equiv \neg(\neg\varphi \vee \neg\psi) && (1)\\
\varphi \to \psi &\equiv \neg\varphi \vee \psi && (2)\\
\mathbf{G}\,\varphi &\equiv \neg(\mathbf{F}\,\neg\varphi) && (3)\\
\varphi\,\mathbf{V}\,\psi &\equiv \neg(\neg\varphi\,\mathbf{U}\,\neg\psi) && (4)\\
\forall_p x : \varphi &\equiv \neg(\exists_p x : \neg\varphi) && (5)
\end{aligned}
$$

Equipped with this semantics, the runtime properties 1–3 in Section 2 can be expressed as LTL-FO$^+$ formulæ. For example, Runtime Property 3 becomes the following:

$$\mathbf{G}\,(\forall_{\text{action}}\, x_1 : x_1 = \text{enter} \rightarrow \forall_{\text{car}}\, x_2 : \\ \mathbf{X}\,(\neg(\exists_{\text{action}}\, x_3 : (x_3 = \text{enter} \wedge \exists_{\text{car}} x_4 : x_2 = x_4))\ \mathbf{U} \\ (\exists_{\text{action}}\, x_5 : (x_5 = \text{exit} \wedge \exists_{\text{car}}\, x_6 : x_2 = x_6))))$$

It states that globally along the message trace, for every message whose action is "enter" and for every car in that message, in the next state the following holds: no message can be an "enter" containing that car until an "exit" message has been observed with the car in question. Remark that quantification is required, since there can be multiple cars in a single "enter" or "exit" message. The Runtime Properties 1 and 2 can be similarly translated into LTL-FO$^+$ formulæ.

3.3 Comparison to Related Work

A number of approaches to the runtime monitoring of systems in general have been suggested over the years. In line with the current concern of the paper to model data dependencies between messages, these approaches can be roughly divided into two categories according to the degree of "data-awareness" they offer. A similar classification was presented in [16] for works concentrating on *static verification* of web services.

3.3.1 Propositional Runtime Monitoring

A first category includes *propositional* runtime monitoring tools, where the sequence of messages is analyzed, but the content of messages is abstracted away. Conversation specifications [6] are an early example of this approach applied to web services; however, these conversations were analyzed under the angle of static verification, and not runtime monitoring.

More recently, propositional runtime monitoring was used by [10], where patterns of messages exchanged by a web service are specified using UML 2.0 Sequence Diagrams, and then transformed into classical finite-state automata whose state is updated for each message sent or received. [19] uses UML Message Sequence Charts with the same intent; however, the authors suggest the application of aspect-oriented programming to call monitoring methods with the use Java *pointcuts*. Different patterns of classical LTL properties were studied by [22], which introduces the concept of *obstacle* to detect possible violations of a specification.

Finally, in [2] an elegant framework for the automatic synthesis of monitors is presented. The language suggested by the authors is called Run-Time Monitor specification Language (RTML), which is an extension of LTL that allows the expression of Boolean (true/false) and numerical properties which can count, for example, the number of times a given message type is received. The content of messages can be statically referred in the properties, but no quantification is allowed on data fields.

3.3.2 Runtime Monitoring with Data Parameterization

There exist a couple of approaches to runtime monitoring which allow some form of quantification on data fields. For example, [3] describes an algorithm for rule-based runtime monitoring, where the rules are temporal fixpoint functions that can include data arguments. [25] makes a similar use of data parameterization for a quantified variant of LTL.

However, these approaches are *state-based*: it is possible to quantify over the state variables of a system, which are only partially equivalent to message contents. Each state variable can take one possible value at every state along the trace. This is different from the quantification over message content used by LTL-FO$^+$. A similar remark applies to the Input-Output State Transition Systems (IOSTS) used in [8] to specify monitoring properties. Although an IOSTS models input and output, the data parameters are not named and therefore cannot be quantified. Hence, in all these works, there is no direct equivalent for the domain function Dom in Definition 4.

More closely related, [14] suggests a framework in which correlations between data in multiple messages are expressed and can be checked at runtime. To the best of our knowledge, the correlations imply a single request-response and do not involve messages arbitrarily far apart in time. Moreover, existential quantification on data fields is not supported.

An alternative to state machines and temporal logics is the use of event calculus (EC), as is done in [21]. The event calculus is a rich extension over first-order logic which allows the expression of constraints over time intervals, in addition to arbitrary predicates over data fields. The semantics of LTL-FO$^+$ could clearly be encoded by a set of EC predicates. However, the richness of the language raises concerns about its applicability in real-world scenarios, as the experiments in [20] suggest. In this respect, we will show that a simpler logic such as LTL-FO$^+$, although less expressive, can be more easily used in concrete contexts.

The recent advances in artifact-centric modelling of business processes led to the development of a logic called ABSL [12]. This logic is an extension of CTL that includes a form of first-order quantification. However, it is suited to express properties of *intra*-artifact behaviours, not *inter*-message constraints; moreover, the approach is not aimed towards runtime monitoring, but rather on static analysis [11]. This is also true of a logic called LTL-FO studied in [9]. The underlying model on which LTL-FO is defined is richer that LTL-FO$^+$'s messaging model; therefore, many problems expressed in LTL-FO are undecidable.

Finally, a different approach has been proposed with specifications using XQuery on traces (SXQT) [26], in which a trace of XML messages is analyzed by means of temporal formulæ converted into XQuery expressions. However, one has to wait for a trace to be complete for the corresponding XML structure to be generated; therefore, this method needs adaptations to be used in a context where the monitoring should occur in parallel with the execution of the workflow.

4 A Runtime Monitor for LTL-FO$^+$

We now describe an algorithm that allows for the runtime monitoring of LTL-FO$^+$ formulæ. We construct a *watcher* for a formula φ which, when fed with the messages from a trace one by one, updates its state and warns of eventual violations of φ.

Definition 7 (Watcher)**.** *A* watcher *for a formula φ is a tuple $\mathcal{W}_\varphi = \langle Q, q_0, \delta, O, f\rangle$ where: Q is a set of* states*; $q_0 \in Q$ is the initial state; $\delta : Q \times M_k \to Q$ is the transition or* update *function; O is a set of* outcomes*, i.e. the possible conclusions that a watcher can draw on a given trace; $f : Q \to O$ is an outcome function.*

Formally, a watcher is a special case of finite-state automaton where the set of accepting states is replaced by a function f returning an "outcome" for each state. The watcher starts in its initial state q_0; then, for each message m that is monitored, the update function $\delta(q, m)$ is called to take the watcher into its updated state S'. At any time during the monitoring process, the outcome function f can be applied on the watcher's state to decide whether the monitored property is violated, fulfilled, or if nothing can yet be concluded from the execution up to that point. This definition makes no assumption about any process instrumentation or annotation. The watcher can be implemented as a local process intercepting messages sent and received, called by aspect-oriented "pointcuts" [19], or implemented as an external observer acting as a verifying layer between acting parties [2].

Although LTL-FO$^+$ is similar in many respects to CTL-FO$^+$, the model checking algorithm developed in [17] cannot be adapted for runtime monitoring. Its effectiveness relies on the fact that data quantification can be modelled as a particular form of branching path quantification. Since LTL-FO$^+$ provides no such path quantifiers, a whole new algorithm must be provided to tackle runtime monitoring.

4.1 Transition Function

The algorithm is inspired from [13] and adapted to the first-order quantification mechanism of LTL-FO$^+$. It is based on the principle that the standard LTL temporal operators can be represented through a fixpoint notation connecting the current and the next state of the trace. For example, the identity $\mathbf{F}\,\varphi \equiv \varphi \vee \mathbf{X}\,(\mathbf{F}\,\varphi)$ indicates that checking $\mathbf{F}\,\varphi$ on a message amounts to checking if φ is true in the current message, and if not, wait for the next state and check $\mathbf{F}\,\varphi$ again. Based on that observation, a simple update algorithm can be developed to keep track of what must be true now, and what must be true in the remainder of a trace.

To this end, we define a *node* as a pair $N = \Gamma \Vdash \Delta$, where Γ is a set of LTL-FO$^+$ formulæ that must be true in the current state, and Δ is a set of LTL-FO$^+$ formulæ that must be true in the next state. We assume without loss of generality that negations can be pushed down to the ground terms by use of identities (1)–(5) and the formulæ $c_1 \neq c_2 = \neg(c_1 = c_2)$ and $\mathbf{X}\,\varphi = \neg(\mathbf{X}\,\neg\varphi)$.

The *state* $q \in Q$ of the watcher consists of a (finite) set of nodes. Intuitively, each node in the watcher's state represents one possible way in which the observed trace can fulfil the property φ. Therefore, the initial state q_0 of $\mathcal{W}_\varphi$ is composed of the single node $\emptyset \Vdash \{\varphi\}$ —that is, no message has yet been observed, and the LTL-FO$^+$ formula φ must hold on the next (i.e. the first) message of the trace. Then, each time a new message m is observed, the state of the watcher is updated via a the δ function shown in Table 1.

The update function simply takes each node $N \in q$, moves the contents of the right-hand side of the node to the left-hand side (leaving the right-hand side empty), and then calls an auxiliary function SPAWN on that resulting node. SPAWN takes a node and decomposes the formulæ from its left-hand side according to the rules shown in Table 2. On some occasions, the decomposition of a formula produces more than one node; the decomposition is then recursively repeated on each resulting node until no further rule applies. The set of these terminal, "spawned" nodes is then returned to δ and included in the new state for the watcher.

Intuitively, the function SPAWN decomposes and evaluates all the LTL-FO$^+$ formulæ that must be true in the current state, eventually evaluating quantified variables and re-

$\delta(m, q)$
  $S' = \emptyset$
  For each $N = \Gamma \Vdash \Delta \in q$
    $N' = \Theta \Vdash \emptyset$
    $S' = S' \cup \textsc{Spawn}(m, N')$
  End for
  Return S'
End function

Table 1. The function δ changes the state of the watcher based on a message observed in the trace.

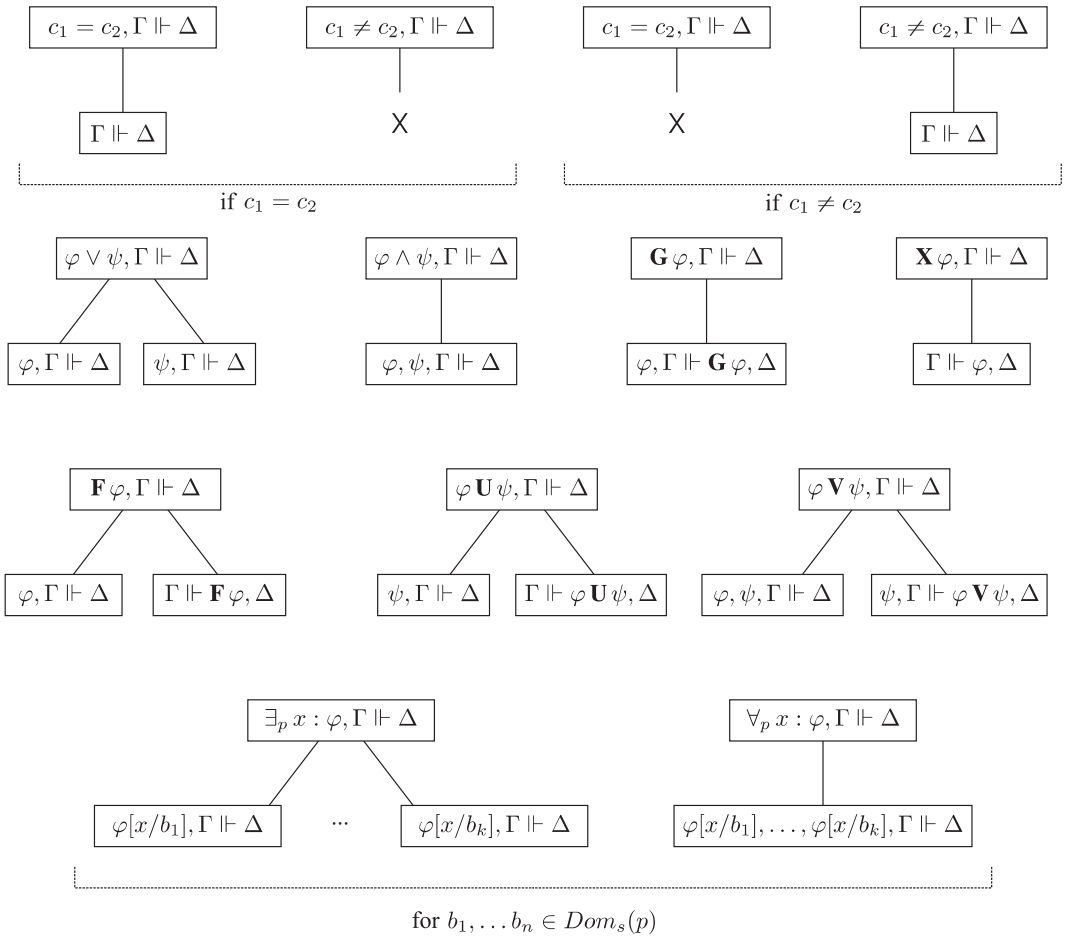

Table 2. The decomposition rules for a watcher's state node. The "X" symbol indicates that the branch is stopped and should be discarded.

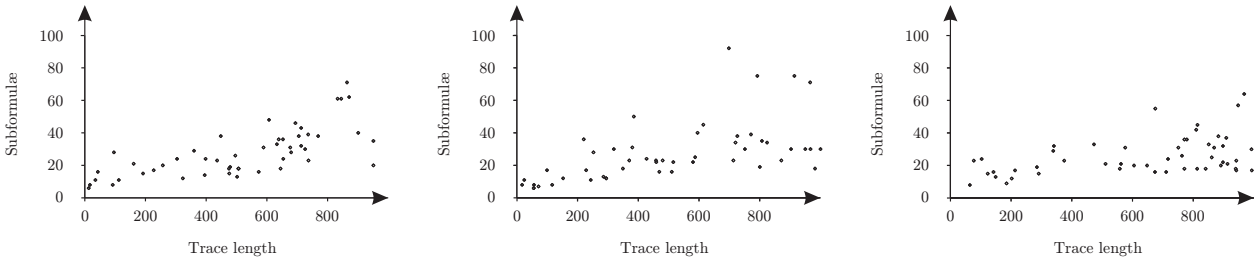

Figure 1. Maximum number of subformulæ in watcher state for various trace lengths, with data domains of respectively 100, 300 and 500 elements.

placing equalities with their Boolean value. At the same time, SPAWN transfers to the right-hand side of the node all the properties that will have to hold in the next iteration of the update function. The number of nodes spawned by the application of a single rule is bounded by the number of elements returned by the function Dom, i.e. k. The resulting tree is therefore of arity at most k. As usual, for a *finite* trace $\rho = \rho_1\rho_2 \dots \rho_n$, we define $\delta(q, \rho) = \delta(\delta(\dots\delta(\delta(q, \rho_1), \rho_2)\dots), \rho_n)$.

4.2 Acceptance Conditions

It remains to determine how the watcher can conclude that a trace fulfils or violates a property. The first case to consider is trace violation:

Definition 8 (Violation condition)**.** *A finite trace $\rho = \rho_1 \dots \rho_n$ violates φ if and only if $\delta(q_0, \rho) = \emptyset$.*

Hence, if a call to UPDATE produces no nodes, then there is no possible way for the trace to continue while still fulfilling the property, and a violation can be announced.

Conversely, the acceptance condition expresses the fact that for a trace to respect the property, it suffices that one of the possible nodes indicates that the property is sure to be true. This is the case when both Γ and Δ are empty: in such a situation, everything that must be true for the current message has been checked, and nothing more needs to be verified when the next message is observed:

Definition 9 (Acceptance condition)**.** *A finite trace $\rho = \rho_1 \dots \rho_n$ fulfils φ if and only if $\emptyset \Vdash \emptyset \in \delta(q_0, \rho)$.*

From these two conditions, the outcome function f can be easily defined. Two possible outcomes of that function are labelled "fulfils" and "violates". However, since most observed traces will be finite, it is possible that the watcher comes to a state where none of the previous conditions apply, although no further message is coming. At this point, since φ was neither confirmed nor violated, then the result is inconclusive.

A more subtle conclusion can be obtained by looking at the actual formula that needs to be checked. For example, a formula of the form $\mathbf{G}\,\varphi$ must be true for all messages of the trace; since no more message is expected, the property is vacuously true and can therefore be considered "not yet violated" by the trace. On the opposite, the temporal operator $\mathbf{F}\,\varphi$ requires that there *exists* a future message such that φ is true; since the trace is completed, no such future message will appear and the property is "not yet fulfilled". A further discussion on the interpretation of LTL formulæ on finite traces can be found in [4]; the discussion could be adapted to LTL-FO$^+$ as well.

The soundness and completeness of this algorithm cannot be proved here due to lack of space; however, the results are based on the proofs presented in [13] for LTL runtime monitoring.

5 Results and Discussion

To evaluate the performance of this runtime algorithm, we conducted a set of initial experiments that involved the runtime monitoring of LTL-FO$^+$ formulæ on automatically-generated traces. The goal of these experiments was to show that the monitoring of LTL-FO$^+$ formulæ can be effectively done in concrete contexts and imposes a reasonable overhead on the execution of a workflow.

5.1 Experimental Results

We chose the Runtime Property 3 because it was the most complex of the three described in Section 2. 50 traces of lengths ranging from 10 to 1,000 messages were randomly produced. Each message consisted in one or more cars either entering or leaving a parking. Each of these traces was then assigned to an instance of a LTL-FO$^+$ runtime monitor following the algorithm described in the previous section. For each run, the total processing time and the maximum number of subformulæ that needed to be kept by the watcher were recorded. The results are presented in the scatterplots of Figure 1 and 2.

A first observation that can be made on these results is that globally, the time required to process a message remains well under 100 milliseconds. All but a dozen traces required less than 40 milliseconds per message to be monitored. These times were computed for a runtime monitor the authors programmed in Java and running on an AMD Athlon XP 2200+ system under Cygwin.[1] The number of subformulæ that needed to be stored by the watcher, and hence the memory footprint of the algorithm, remains within reasonable bounds and grows proportionnally to the trace length.

To measure the influence of the data domain size on the performance, the experiments were repeated for domains of 100, 300 and 500 cars. We could not conclude that the size of the data domain has a decisive impact on the performance of the runtime monitoring algorithm. In this respect, our results confirm what was observed by [20] for the runtime monitoring of event calculus formulæ. We believe the same explanations given in that paper apply here.

These initial findings are encouraging and suggest that runtime monitoring of LTL-FO$^+$ can be performed in real-world scenarios, and that augmenting existing specification languages with quantification over data does not impose a large overhead on their processing.

[1] The files used for the experiments are available online at http://teleinfo.uqam.ca/Members/halle_sylvain/watcher.

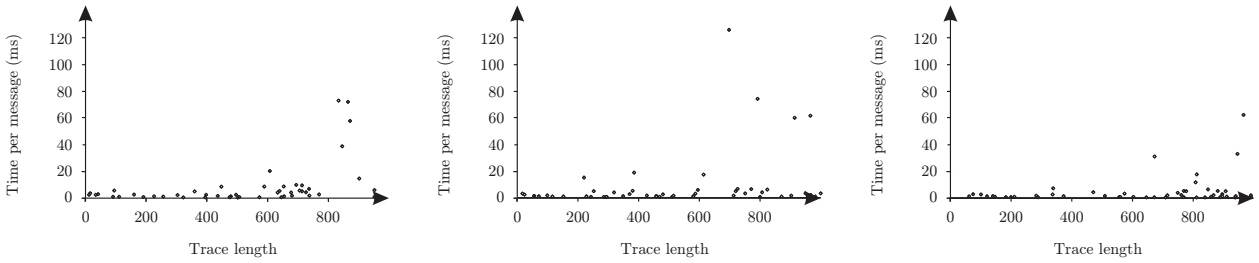

Figure 2. Processing time per message for various trace lengths, with data domains of respectively 100, 300 and 500 elements.

5.2 Further Refinements

A number of adaptations to the original approach can be made to support a wider range of monitoring properties. We briefly mention two of them.

5.2.1 Nested messages

The extension of LTL-FO$^+$ runtime monitoring to arbitrary XPath expressions on messages is straightforward. The interpretation of the function $Dom_s()$ can be arbitrary, as long as it returns a set of values. Therefore, instead of a single parameter name, an arbitrary XPath expression could be entered and evaluated by the monitor by calling an external XPath engine on the received message. It then suffices to replace the subscript in a quantifier by the appropriate XPath expression. A formula like $\exists_{/client/customerID}\varphi$ would indicate that φ holds for every "customerID" found under the "client" element of a message.

5.2.2 Metric Temporal Logic

Metric temporal logic (MTL) is an extension of regular temporal logic to time intervals. Time intervals are used for expressing time delays in business contracts, as the properties in [15] demonstrate. MTL formulæ are also used in [1] to express monitoring properties on plan execution for NASA's planetary rover controller K9. For example, an expression like $\mathbf{G}\,(start(plan) \rightarrow \mathbf{F}_{1,5}\,start(drive1))$ indicates that if the `plan` starts, then task `drive1` should begin execution within 1 and 5 time units.

Such constraints can be simulated by simply adding a timestamp τ to each message. The actual timestamp need not even be exchanged through messages, but quantification on τ could simply amount to fetching the current timestamp from an internal clock. In the same way as [3], metric temporal logic then becomes a particular case of data parameterization. The previous property could hence be translated into LTL-FO$^+$ as follows:

$$\mathbf{G}\,(start(plan) \rightarrow (\forall_\tau t_1 : \mathbf{F}\,(start(drive1) \wedge \forall_\tau t_2 : 1 \leq |t_2 - t_1| \leq 5)))$$

6 Conclusion

In this paper, we have presented an algorithm for the runtime monitoring of data-aware workflow constraints. Sample properties taken from runtime monitoring scenarios in existing literature were expressed using LTL-FO$^+$, an extension of Linear Temporal Logic that includes full first-order quantification over message contents. An on-the-fly runtime monitoring algorithm was presented and tested on sample traces. Initial findings indicate that runtime monitoring of LTL-FO$^+$ can be performed in real-world scenarios for traces of a thousand messages while still allowing for quantification over data domains of 500 elements. Further experiments in various contexts will be conducted; a possible extension of this work involves the development of mappings between LTL-FO$^+$ and patterns in graphical languages such as UML.

References

[1] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. R. Lowry, C. S. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, pages 87–107. Springer, 2003.

[2] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS*, pages 63–71. IEEE Computer Society, 2006.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and

G. Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[4] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In Sokolsky and Tasiran [24], pages 126–138.

[5] R. Boutaba, W. Golab, Y. Iraqi, and B. S. Arnaud. Lightpaths on demand: A web-services-based management system. *IEEE Communications Magazine*, pages 2–9, July 2004.

[6] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[8] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Trans. Software Eng.*, 33(8):558–574, August 2007.

[9] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven web services. In S. Vansummeren, editor, *PODS*, pages 90–99. ACM, 2006.

[10] Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O'Farrell, and J. Waterhouse. Runtime monitoring of web service conversations. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 42–57, New York, NY, USA, 2007. ACM.

[11] C. E. Gerede, K. Bhattacharya, and J. Su. Static analysis of business artifact-centric operational models. In *SOCA*, pages 133–140. IEEE Computer Society, 2007.

[12] C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In B. J. Krämer, K.-J. Lin, and P. Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2007.

[13] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.

[14] C. Ghezzi and S. Guinea. *Run-Time Monitoring in Service-Oriented Architectures*, pages 237–264. Springer, 2007.

[15] G. Governatori, Z. Milosevic, and S. W. Sadiq. Compliance checking between business processes and business contracts. In *EDOC*, pages 221–232. IEEE Computer Society, 2006.

[16] S. Hallé, R. Villemaire, O. Cherkaoui, and B. Ghandour. Model-checking data-aware temporal workflow properties with CTL-FO+. In *EDOC*, pages 267–278. IEEE Computer Society, 2007.

[17] S. Hallé, R. Villemaire, O. Cherkaoui, J. Tremblay, and B. Ghandour. Extending model checking to data-aware temporal properties of web services. In M. Dumas and R. Heckel, editors, *WS-FM*, volume 4937 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.

[18] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[19] I. H. Krüger, M. Meisinger, and M. Menarini. Runtime verification of interactions: From MSCs to aspects. In Sokolsky and Tasiran [24], pages 63–74.

[20] K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS*, pages 257–265. IEEE Computer Society, 2005.

[21] K. Mahbub and G. Spanoudakis. *Monitoring WS-Agreements: An Event Calculus-Based Approach*, pages 265–306. Springer, 2007.

[22] W. Robinson. A requirements monitoring framework for enterprise systems. *Requir. Eng.*, 11(1):17–41, 2006.

[23] W. N. Robinson. Monitoring web service requirements. In *RE*, pages 65–74. IEEE Computer Society, 2003.

[24] O. Sokolsky and S. Tasiran, editors. *Runtime Verification, 7th International Workshop, RV 2007, Vancover, Canada, March 13, 2007, Revised Selected Papers*, volume 4839 of *Lecture Notes in Computer Science*. Springer, 2007.

[25] V. Stolz. Temporal assertions with parametrised propositions. In Sokolsky and Tasiran [24], pages 176–187.

[26] M. Venzke. Specifications using XQuery expressions on traces. *Electr. Notes Theor. Comput. Sci.*, 105:109–118, 2004.