Runtime Monitoring of Web Service Choreographies Using Streaming XML

Sylvain Hallé*
University of California, Santa Barbara
Department of Computer Science
Santa Barbara, CA 9310-65110
shalle@acm.org

Roger Villemaire
Université du Québec à Montréal
C.P. 8888, Succ. Centre-ville
Montreal, Canada H3C 3P8
villemaire.roger@uqam.ca

ABSTRACT

A wide range of web service choreography constraints on the content and sequentiality of messages can be translated into Linear Temporal Logic (LTL). Although they can be checked statically on abstractions of actual services, it is desirable that violations of these specifications be also detected at runtime. In this paper, we show that, given a suitable translation of LTL formulæ into XQuery expressions, such runtime monitoring of choreography constraints is possible by feeding the trace of messages to a streaming XQuery processor. The forward-only fragment of LTL is introduced; it represents the fragment of LTL supported by available streaming engines.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*temporal logic*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*web-based services*; I.7.2 [**Document and Text Processing**]: Document Preparation—*XML*

General Terms

Theory, verification

Keywords

Runtime monitoring, web services, streaming XML

1. INTRODUCTION

A web service choreography specification can loosely be seen as a set of constraints on the messages exchanged by web services. Since most web services exchange XML messages, one can refer to a recorded trace as an XML "document" that can be analyzed using standard XML tools, using for example the XML Query Language (XQuery) [19,28]. However, most works attempting to tap on the resources available in XQuery engines operate in a *post mortem* fashion: an instance of a choreography must be finished before analysis can take place on a complete XML document. While in some cases, a *post mortem* analysis on recorded traces is appropriate, there exist situations where violations of a specification must be addressed as soon as they are discovered.

The recent years have seen the rise of a new form of XML evaluation called *streaming XML*: an XML stream is a linear sequence of events generated by the parsing of a document. Streaming XML engines evaluate XPath or XQuery expressions by taking as input only that stream of events. These engines have been developed with the intent of processing very large XML documents that could not fit in the memory of traditional tools. In this paper, we show how streaming XML engines can be used for a second, perhaps unexpected purpose, by turning them into runtime monitors for web service interactions. In Section 2, we show by means of a simple e-commerce example how a trace of XML messages exchanged by web services can be fed progressively to an XQuery engine, which reads and processes it as a streaming "document".

This principle is formalized in Section 3, where we recall how Linear Temporal Logic can be used to express a wide range of interaction properties. We then provide a mapping between LTL and XQuery expressions suitable for streaming XML, and study some extensions of LTL that can also be translated into XQuery at no additional cost.

We put a special care in porting our runtime monitoring approach to a practical context; this is why in Section 4, we survey readily-available XQuery engines with respect to their streaming capabilities and show that most existing products cannot be used to monitor full-fledged Linear Temporal Logic. We introduce LTL$^{\rightarrow}$, the *forward-only* fragment of LTL, which corresponds to the "least-common denominator" that can be monitored on engines with limited streaming capabilities. Experimental findings indicate that the use of a streaming XQuery engine to perform runtime monitoring of LTL$^{\rightarrow}$ can be implemented with negligible additional load on existing web service execution environments, while providing for immediate, real-time detection of a substantial class of choreography violations.

*This work was done when Sylvain Hallé was at Université du Québec à Montréal. We gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada on this research.

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.
SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

2. MOTIVATION

In this section, we provide an example of a web service scenario where choreography constraints must be monitored at runtime. In particular, we focus on the subset of choreographies which constrain the sequence of messages exchanged by one specific partner with its peers. This form of interaction can be seen as a "contract" to be locally monitored and enforced. Assuming that web services interact by exchanging structured XML messages, as is the case for a substantial portion of existing resources, we then show informally how streaming XML engines can be used to perform that monitoring.

2.1 Runtime Monitoring of Web Services

Consider an e-commerce scenario where a shop offers users to buy products through a web service interface. A client service first logs into the system by providing a user name. The shop offers a discount if a user connects with the commitment to buy at least one product, which is signalled with the `commitToBuy` element. The shop responds to the login with a `loginConfirmation`, providing a unique ID for the session. Additionally, if the user's commitment to buy a product has been accepted by the shop, a maximum delay in minutes before which the first transaction must take place is given in the `expiration` element.

The user can then retrieve the product list, ask for details about specific products, and eventually send a `buyOrder` message indicating the products and respective quantities that it wishes to buy. Several such messages can be sent, and the transaction is concluded by sending a `confirmPayment` message, providing as a safety measure the unique ID given by the shop at the start of the session.

There exist several interaction constraints that should be monitored at runtime under such a scenario. For example, from the online shop point of view, one might want to make sure that users committing to buy actually do so eventually.

CONTRACT SPECIFICATION 1. *A user whose commitment to buy has been accepted by the shop will eventually send a* `buyOrder` *message.*

To prevent clients from buying products with outdated information, the shop might require that no `buyOrder` be sent before first asking for the latest product listing:

CONTRACT SPECIFICATION 2. *No* `buyOrder` *can be sent before first receiving a product list.*

To make sure that the safety measure is enforced, one might also want to monitor that the unique session ID sent at the start of the session be present in every confirmation.

CONTRACT SPECIFICATION 3. *The same unique ID must be present in all confirmation messages.*

The shop can also monitor that clients that commit to buy actually do so before the timeout sent by the shop:

CONTRACT SPECIFICATION 4. *The first* `buyOrder` *of a client whose* `commitToBuy` *has been accepted by the shop must occur within x minutes of the login, with x the timeout value sent when the session was opened.*

2.2 In-memory vs. Streaming XML

There exist two main modes of representing and processing XML documents by XPath and XQuery engines. In the *Document Object Model* (DOM) [2], the nested tag structure of the original document is translated in memory into a tree model; to process a query, the engine can retrieve arbitrary parts of of the document by specifying the path to the desired nodes. Although relatively straightforward, this method suffers from the fact that the whole document must be loaded in memory in order to be processed. Experimental benchmarks showed that, as a consequence, DOM-based XQuery engines are unable to process large documents [22].

An alternate approach consists of representing an XML document as a sequence of events generated as it is parsed. Only this sequence of events is fed to the engine, which consumes them in their order of arrival and updates its state to compute the desired query result.

Streaming XML shifts the processing burden from the parser, which is relieved from building a tree structure from XML code, to the query engine, which is forced to update its state based on a linearized version of the document. Developing such an engine is generally more complex than for a DOM-based solution: it involves tracking the nesting of elements and carefully memorizing whatever parts of the document are required to compute a result. In counterpart, because streaming query engines do not require the whole document to be loaded at once in memory, they can process documents orders of magnitude larger than DOM engines.

2.3 Monitoring With Streaming XML

A second, crucial advantage of streaming XML is that the query results are also streamed: whenever possible, the engine sends its results progressively to an output pipe, while the input document is being read, and without having to wait until the end of document has been reached. This feature of streaming XML can be put to good use to perform basic monitoring of web service choreographies at runtime.

For example, suppose a sequence of messages $M_1, M_2, \ldots$ is streamed to an XQuery engine as the following "document": <trace> <message> M_1 </message><message> $M_2 \cdots$ < /message> $\cdots$ < /trace>, where M_i is the XML body of the i-th message. Then consider the following expression Q:

```
every $x in /trace/message[2] satisfies
        (not($x/commitAccepted = "true") or
      some $y in $x/following-sibling::* satisfies
                              $y/name = "buyOrder"
```

An XQuery engine computing Q on the above stream will send `true` on its output pipe if the second message of the trace does not contain an element `commitAccepted` with a value of "true", or otherwise as soon as a `buyOrder` message is encountered. If the stream is closed by reading the </trace> element, `false` is sent on the output pipe.

Based upon that observation, a streaming XQuery processor can be used as a crude web service runtime monitor: as shown in Figure 1:

1. From a monitoring constraint, build an XQuery query Q such that Q returns `false` if and only if the constraint is violated

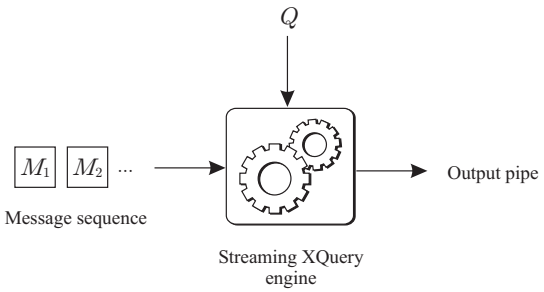

Figure 1: Runtime monitoring through streaming XML

2. Compute the result of Q with a streaming XQuery engine on a "document" formed by progressively streaming the messages exchanged by the service

3. Raise an error as soon as false is read on the output pipe; stop monitoring as soon as true is read on the output pipe.

The use of XQuery engines to perform various forms of validation and monitoring on web services has been little studied. [8] suggested the implementation of web service stubs that intercept in- and outbound messages and validate structural properties expressed in a language called CLiX. In that framework, every message is validated independently of each other; there is no streaming involved and properties like expression Q above are out of reach of the approach. The validation of web service message traces with XQuery was sketched in [19,28]. However, the approach only applied to the *post mortem* validation of recorded traces; moreover, it was restricted for [28] to special cases of Hoare's event-condition-action and request-response patterns, and not to a full temporal logic as it is done in this paper. In both cases, the resulting XQuery expressions cannot be evaluated by existing XQuery engines in streaming mode. Therefore, as far as the authors could check, this work is the first application of the *streaming* capabilities of XQuery engines to perform *runtime* monitoring of web service message traces.

The motivation for using XQuery engines for runtime monitoring is to leverage existing resources available in production environments. Many existing solutions for runtime monitoring require changes to the web service execution environments; for example, runtime monitors in [5, 20] require the implementation of their own monitoring algorithms. In contrast, most web service execution environments provide XML processing capabilities natively, thus providing a minimally intrusive way of adding a runtime monitoring capability with existing technologies.

3. FORMALIZATION

We now systematize the approach suggested above by rephrasing it into a formal model. More specifically, we show how the web service constraints of Section 2.1 can be expressed in Linear Temporal Logic (LTL), and then how LTL formulæ can be mapped into XQuery expressions suitable for streaming processors.

3.1 Linear Temporal Logic

We start by briefly recalling how Linear Temporal Logic (LTL) can be used to formally express the previous interaction requirements. LTL has been introduced to express properties about states and sequences of states in systems called Kripke structures [11]. In the present case, the states to be considered are messages inside a conversation. A sequence of messages $M_1, M_2, \ldots$ is called a message trace.

The basic units of LTL formulæ are called *ground terms*; in the present case, a ground term is any path expression $\pi =$ "d" on a single message that evaluates to true when the value at the end of π is equal to d. LTL formulæ are built up from ground terms and the constants *true* and *false* using the classical connectors: $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies) and $\neg$ (not). LTL further provides *temporal operators* that can be used on top of traditional propositional logic formulæ to specify conditions on the ordering of the messages.

The first of these operators is **G**, which means "globally". For example, the formula $\mathbf{G}\,\varphi$ means that formula φ is true in every message from now on. The operator **F** means "eventually"; the formula $\mathbf{F}\,\varphi$ is true whenever φ holds for some future message. The operator **X** means "next"; it is true whenever φ holds in the next message. Finally, the **U** operator means "until"; the formula $\varphi\,\mathbf{U}\,\psi$ is true if ψ eventually holds, and until then φ holds for all messages. The "weak until" **W** behaves like **U** but drops the requirement that ψ eventually holds.

Definition 1 (LTL syntax). *A formula φ belongs to LTL if and only if it is formed from the following BNF grammar:*

$$\varphi \equiv \neg\varphi | \varphi \vee \varphi | \varphi \wedge \varphi | \varphi \rightarrow \varphi | \mathbf{G}\,\varphi | \mathbf{F}\,\varphi | \mathbf{X}\,\varphi | \varphi\,\mathbf{U}\,\varphi | \pi = \textit{"d"}$$

Equipped with this logic, the web service constraints of Section 2.1 can be formalized in LTL.

LTL Contract Specification 1.

$$(\text{commitToBuy} = \textit{true}) \rightarrow \mathbf{F}\,(\text{name} = \text{buyOrder})$$

This first formula expresses that when the initial message's element `commitToBuy` has value "true", then eventually, some message will have `buyOrder` for its name. Interaction Specification 2 can be similarly formalized:

LTL Contract Specification 2.

$$(\text{name} \neq \text{buyOrder})\,\mathbf{W}\,(\text{name} = \text{productList})$$

Translating web service choreography constraints of various kinds into LTL has been amply covered in the literature; the reader will find in [13, 15] many other examples. The choice for LTL is motivated by its relative closeness to XQuery from a formal semantics standpoint, a property that will be crucial in the next subsection. Moreover, we shall see that other choreography description languages can also be translated into LTL.

3.2 Translating XQuery into LTL

Remark that a message trace $\sigma = \sigma_1\sigma_2\ldots$ can be seen as an XML document by itself. It suffices to encapsulate each message sequentially into a global *trace* element as follows:

```
<trace>
  <message>
    M1
    <message>
      M2
      ...
    </message>
  </message>
</trace>
```

The recursive nesting of the messages within each other is voluntary. We shall see in Section 4 why putting messages side-by-side under the root element, as was shown in Section 2.3, is not an appropriate choice although it seems more natural. Note that this nesting implies that each `message` element has no sibling and at most one direct child.

Throughout the following definitions, we shall repeatedly use the "[1]" construct, which designates in XQuery the first sibling satisfying the path expression. When used for `message` elements, which have no sibling by construction, this has for effect of telling the XQuery engine not to wait until the closing parent tag before returning its result, since only the first `message` element needs to be considered. Otherwise, all the query results would be output to the pipe at the end of the document, which defeats the purpose of runtime monitoring.

We develop a recursive translation function ω_ρ, which takes as input an LTL formula and produces an equivalent XQuery expression. The function also carries a parameter ρ, which is a pointer to the current message of the trace. When starting the translation, ρ must point to the first message of the trace document. The XPath expression /trace/message[1] can be used to designate this first message.

It suffices to define ω_ρ for each of the constructs given in Definition 1. The translation of XML path expressions is direct: for π a path in a message and $d \in D$ a value at the end of that path, the following FLWOR expression gives an equivalent Boolean result:

$$\omega_\rho(\pi = d) \equiv \text{for \$x in } \rho/\pi[1] \text{ return \$x} = \text{"}d\text{"}$$

It is important to remark that the path π is *relative* to the current message of the trace; hence π must be appended to ρ. XQuery allows all logical connectors, therefore, the translation of $\neg$, $\vee$ and $\wedge$ is also straightforward; it has, though, to be encapsulated within a FLWOR expression.

$$\begin{aligned}
\omega_\rho(\varphi \vee \psi) &\equiv \text{for \$x in } \rho \text{ return } (\omega_{\$x}(\varphi) \text{ or } \omega_{\$x}(\psi)) \\
\omega_\rho(\varphi \wedge \psi) &\equiv \text{for \$x in } \rho \text{ return } (\omega_{\$x}(\varphi) \text{ and } \omega_{\$x}(\psi)) \\
\omega_\rho(\neg\varphi) &\equiv \text{for \$x in } \rho \text{ return not}((\omega_{\$x}(\varphi))
\end{aligned}$$

In this translation, $x is a fresh XQuery variable, bound to the every statement. By construction, there is only one candidate for $x; hence the FLWOR expression returns only one Boolean value, and not a sequence of Booleans.

It remains to translate the temporal operators into equivalent XQuery code. We consider first the case of the **G** operator. According to the semantics of LTL, a formula of the form $\mathbf{G}\,\varphi$ is true on the trace which starts at the current message, if and only if all subsequent messages (including the current one), satisfy φ. Since ρ is a pointer to the current message, then the path expression ρ//message[1] denotes all the messages starting from ρ and following it. The XQuery formula must then express that each message in this set satisfies the remaining formula φ, or more precisely, the translation of φ into XQuery. We obtain the following expression:

$$\omega_\rho(\mathbf{G}\,\varphi) \equiv \text{every \$x in } \rho\text{//message[1] satisfies } \omega_{\$x}(\varphi)$$

Remark that since φ must be true on each such message, the root on which φ is evaluated is $x.

The translation of the "next" (**X**) operator, can be seen as a special case of **G**, where the set of desired messages contains only the immediate successor to ρ:

$$\omega_\rho(\mathbf{X}\,\varphi) \equiv \text{every \$x in } \rho\text{/message[1] satisfies } \omega_{\$x}(\varphi)$$

The translation of $\mathbf{F}\,\varphi$ states that some message in the future satisfies φ:

$$\omega_\rho(\mathbf{F}\,\varphi) \equiv \text{some \$x in } \rho\text{//message[1] satisfies } \omega_{\$x}(\varphi)$$

We postpone the translation of the remaining LTL operator, **U**, until Section 4.3.

As an example, applying ω to LTL Interaction Specification 1 yields the following XQuery expression:

```
for $x in /trace/message[1] return
  (for $u in $x/commitToBuy[1] return not($u = "true"))
  or
  (some $y in $x//message[1] satisfies
    (for $v in $y/name[1] return $v = "buyOrder"))
```

An interesting consequence of this mapping is that any other notation translatable into LTL can also be validated using XPath. This includes UML Sequence Diagrams (or more specifically Message Sequence Charts), since various algorithms to translate them into LTL have been developed, for instance in [9, 16]. This also includes SSDL's Message Exchange Patterns (MEP) and Rules protocol frameworks [23], and the *Let's Dance* choreography description language [12].

3.3 Extensions to LTL

The mapping provided above covers a subset of XQuery: there exist valid XQuery expressions which do not correspond to any LTL formula. We can take advantage of the greater expressive power of XQuery to introduce a few extensions to classical LTL, which can be verified by an XQuery engine at no additional cost.

3.3.1 Data Correlations Between Messages

The first of these extensions allows for the expression of *data-aware* correlations. A data-aware web service property is a constraint on the pattern of message exchanges where the order of messages and their data content are interdependent, as was suggested by [20].

For example, Interaction Specification 3 cannot be expressed with classical LTL, for it compares the value of two message elements at two different moments in time. The

only solution, as pointed out by [13], is to include first-order quantification over message elements, yielding a logic called LTL-FO$^+$.

DEFINITION 2 (LTL-FO$^+$ SYNTAX). *Let ψ be an LTL formula. An LTL-FO$^+$ formula is obtained by the following grammar:*

$$\varphi \equiv \psi | \exists_\pi x : \varphi | \forall_\pi x : \varphi$$

Informally, an expression of the form $\forall_\pi x : \varphi(x)$ states that, for all possible values k taken at the end of path π, $\varphi(k)$ holds. Similarly, $\exists_\pi x : \varphi(x)$ requires φ to hold for one value taken at the end of π.

The translation of first-order quantification in XQuery becomes straightforward.

$$\begin{aligned} \omega_\rho(\forall_\pi x : \varphi) &\equiv \text{every \$x in } \rho/\pi \text{ satisfies } \omega_\rho(\varphi) \\ \omega_\rho(\exists_\pi x : \varphi) &\equiv \text{some \$x in } \rho/\pi \text{ satisfies } \omega_\rho(\varphi) \end{aligned}$$

It is now possible to express Interaction Specification 3 in LTL-FO$^+$:

LTL-FO$^+$ CONTRACT SPECIFICATION 3.

$$\mathbf{X}\, \exists_{\text{uniqueID}} x : \\ \mathbf{G}\, (\text{message} = \text{confirmPayment} \rightarrow \\ \exists_{\text{uniqueID}} y : x = y)$$

3.3.2 Metric Temporal Logic

Metric temporal logic (MTL) is an extension of regular temporal logic for expressing time delays in business contracts [3,18]. For example, Interaction Specification 4 specifies that some event (the first `buyOrder` message) must occur before some timeout whose value is specified dynamically at runtime.

Such constraints can be handled by allowing formulæ to access the value of a global clock τ provided by the execution environment. A quantification on τ simply amounts to fetching the current timestamp from that internal clock; metric temporal logic then becomes a particular case of data parameterization. Interaction Specification 4 can hence be translated into LTL-FO$^+$ as follows:

LTL-FO$^+$ CONTRACT SPECIFICATION 4.

$$\mathbf{X}\, (\text{commitAccepted} = \mathit{true} \rightarrow \\ \forall_\tau t \forall_{\text{expiration}} k : \mathbf{F}\, (\text{name} = \text{buyOrder} \\ \wedge \forall_\tau t' : t' - t < k))$$

4. PORTING TO EXISTING ENGINES

As was mentioned, evaluating XQuery expressions in streaming mode is generally more difficult to implement than for DOM. The efficient evaluation of XQuery on streaming XML is still an open problem subject to a large amount of research; the reader is referred to [21, 22, 24] for a sample of relevant works on that topic. Consequently, the support for XQuery in streaming mode is still partial and varies greatly from tool to tool.

One can observe that the structure of the trace file shown in the motivating example of Section 2.3 is different from the one we retained in our formal presentation in Section 3.2: the former aligns all messages as siblings under the root `trace` element, while the latter nests them as descendants. Consequently, the formal translation of Interaction Specification 1 is also different from the example given at the beginning of this paper. Although both methods are logically equivalent, only the second can be evaluated by streaming engines. No XQuery tool currently supports the `following-sibling` and `previous-sibling` axes in streaming mode.

Therefore, we must take care that the translation we provide be supported by actual XQuery engines, and in particular the following three features: quantifiers some and every, descendant (//) axis and sibling position ([1]) function. In this section, we survey the streaming capabilities of currently available engines with respect to these features and adapt our methodology to make the best out of them.

4.1 Available Streaming Capabilities

The W3C maintains a list of XQuery implementations[1] which can be classified into three categories.

4.1.1 Insufficient streaming support

In this category fall all XQuery engines that must be discarded for various reasons:

- Some do not support streaming XML processing at all: these include Galax [27] and XMLTaskForce [17]
- Some XQuery engines support only XPath 1.0 in streaming mode, such as TwigM [10], XSQ [25], TurboXPath [4] and XAOS [6].
- Some streaming XQuery engines only support a fragment of the language: GCX [26] does not support the quantifiers every and some, Nux[2] forbids the use of the descendant axis.

4.1.2 Sufficient streaming support

In this category, we find the XQuery engines that provide full streaming support for all the language features required by our translation. Tools in this category are mostly academic and experimental, such as XQPull [14] and MXQuery [7]. Both provide full access to the source code.

4.1.3 Partial streaming support

Although they support streaming processing of XQuery expressions with quantifiers and descendant axes, the tools in this category impose restrictions on the way in which they can be used.

By definition, a streaming XML source can only be read in the forward direction, and this can be done only once. This entails that any XQuery result that requires some form of backtracking in the source document cannot be handled. Fully streaming engines circumvent this problem by carefully memorizing the parts of the stream which will need to be used later in the computation of a result, so that rewinding in the source is not required. On the contrary, "partial" engines perform no memorization and cannot evaluate some queries in streaming mode, even though they are formed of language constructs supported by the streaming engine.

[1] http://www.w3.org/XML/Query/#implementations
[2] http://dsd.lbl.gov/nux

Although a few tools fully support our translation of LTL to XQuery, engines with partial support are more representative of the streaming capabilities likely to be found in an actual web service execution environment. The remainder of this paper therefore concentrates on tools from this category. Notable proponents include Saxon[3] and DataDirect XQuery[4] (DDXQ), two commercial products that we tested under an evaluation license.[5]

4.2 The Forward-Only Fragment of LTL

We study in this section the fragment of LTL that can be supported within the limits of XQuery engines with partial streaming support. We call it the *forward-only fragment* of LTL, noted LTL$^{\rightarrow}$, since it corresponds to formulæ which can be evaluated without backtracking in the message trace. We shall first identify a set of syntactical conditions on the structure of an LTL formula which are sufficient to prevent backtracking.

1. No temporal operator can be in the scope of $\mathbf{G}$. Indeed, evaluating a formula of the form $\mathbf{G}\,\spadesuit\varphi$, with $\spadesuit \in \{\mathbf{G}, \mathbf{F}, \mathbf{U}, \mathbf{X}\}$ on a message trace $M_1, M_2, \ldots$ requires first evaluating $\spadesuit\varphi$ on M_0. However, evaluating the temporal operator $\spadesuit$ will require reading M_1 and possibly M_2, M_3, and so on. Once $\spadesuit\varphi$ has been decided for M_1, it needs to be evaluated again starting at M_2, yet due to the previous evaluation of $\spadesuit\varphi$ on M_1, we can no longer guarantee that the source has not been read past M_2.[6] By a similar reasoning, no temporal operator can be in the scope of $\mathbf{F}$ or $\mathbf{U}$.

2. In contrast, a temporal operator *can* be in the scope of $\mathbf{X}\,\varphi$. The LTL "next" operator simply has for effect of jumping to the next message and evaluating φ from there; no backtracking is needed.

Since temporal operators express properties about the sequence of messages, it is natural that some restrictions apply when a trace can only be read in one direction. More surprisingly however, the forward-only consumption of a message trace also restricts the use of the Boolean connectives:

3. In a formula of the form $\varphi \wedge \psi$, no temporal operator can be in the scope of φ. It suffices to realize that both φ and ψ must be evaluated from the same starting point; therefore, the presence of a temporal operator in φ can possibly consume messages which will need to be rewound when the evaluation of ψ takes places. Remark that the opposite works: if φ does not contain any temporal operator, then the source is still at M_1 when the evaluation of ψ starts, and the source can then be consumed as much as we want. A similar reasoning can be made to forbid temporal operators in the left member of a disjunction ($\vee$). It follows that in the forward-only fragment of LTL, Boolean connectives are not commutative.

4. The negation has no restriction attached to it.

[3]http://www.saxonica.com
[4]http://www.datadirect.com
[5]An open source version of Saxon is also available, but as of July 2008, it did not allow streaming XML processing.
[6]These conditions are sufficient, but not necessary. For example, $\mathbf{G}\,(\mathbf{G}\,\varphi) \equiv \mathbf{G}\,\varphi$ and $\mathbf{G}\,(\mathbf{X}\,\varphi) \equiv \mathbf{X}\,(\mathbf{G}\,\varphi)$.

Finally, when we extend LTL with the first-order quantifiers introduced in Section 3.3, the forward-only fragment of LTL also imposes restrictions:

5. No temporal operator can be in the scope of a universal quantifier. Indeed, evaluating $\forall_\pi x : \varphi$ on a message trace $M_1, M_2, \ldots$ requires checking that φ is respected for all values of x admissible for π. When a first value for x is picked, if φ is contains a temporal operator, then messages $M_1, M_2, \ldots$ might be consumed before deciding on the value (true or false) of φ. Once this is done, a second value for x must be chosen, and φ must be checked again, starting from M_1. As previously, we cannot guarantee that further messages from the source have been consumed. A similar reasoning can be made to forbid temporal operators under an existential quantifier.

6. We can, however, introduce a weaker version of the quantifiers, noted $\exists^1_\pi$ and $\forall^1_\pi$, that assume that, in every message, there exists at most one possible value for the variable. Hence, the quantifier $\exists^1_p x : \varphi$ is true when the value of the single element "p" in the current message satisfies φ. Since no backtracking is involved to decide the quantifier, temporal operators can be present in its scope, provided they respect the above rules.

Based on these limitations, we can now define the BNF grammar of the forward-only fragment of LTL(-FO$^+$) as follows:

Definition 3 (Forward-only fragment). *A formula ψ is in the forward-only fragment of LTL if it is generated by the following grammar.*

$$\begin{aligned}\psi &\equiv \varphi|\mathbf{G}\,\varphi|\mathbf{F}\,\varphi|\varphi\,\mathbf{U}\,\varphi|\varphi\,\mathbf{W}\,\varphi|\mathbf{X}\,\psi|\exists^1_\pi x : \psi|\forall^1_\pi x : \psi| \\ &\quad\ \varphi \wedge \psi|\varphi \vee \psi|\neg\psi \\ \varphi &\equiv \exists_\pi x : \varphi|\forall_\pi x : \varphi|\exists^1_\pi x : \varphi|\forall^1_\pi x : \varphi|\varphi \wedge \varphi|\varphi \vee \varphi|\neg\psi|\pi = \text{“}d''\end{aligned}$$

Specifications 1 and 2 clearly fulfil these restrictions. For Specification 3, the two occurrences of $\exists$ can be replaced by the weakened $\exists^1$, since there is at most one `uniqueID` element in every message. The same applies for the two quantifications over time τ in Specification 4, since there is only one time value at any moment. Finally, the quantification over the `expiration` element can be weakened too, for there is only one such element in the login reply. Therefore, all the Interaction Specifications shown previously belong to LTL$^{\rightarrow}$.

Although restrictive, the forward-only fragment of LTL presents two advantages:

- The logic is *bottom-up*: it is defined from the streaming capabilities of existing engines and therefore constitutes a least common denominator for runtime monitors.

- The logic is *simple*: since no memorization of the document is required, it can be monitored by an XQuery engine in a small and *constant* memory space.

4.3 Mapping the "Until" Operator

Up to now, the translation of the LTL **U** operator in XQuery has not been covered. Unlike the other LTL operators, **U** is more complex, since it asserts two different things: 1) ψ eventually occurs for some message, and 2) for every preceding message, φ holds. A classical result in temporal logic shows that **G** and **F** are particular cases of until; more specifically, $\mathbf{F}\,\varphi \equiv \text{true}\,\mathbf{U}\,\varphi$ and $\mathbf{G}\,\varphi \equiv \neg(\text{true}\,\mathbf{U}\,\neg\varphi)$.

There exist two possible ways of translating $\varphi\,\mathbf{U}\,\psi$ into XQuery. The first one consists in giving an equivalent XQuery expression:

```
some $x in ρ/descendant::*/message satisfies ω_ρ(φ) and
    every $y in (ρ/descendant::*/message intersect
                 $x/ancestor::*/message) satisfies ω_ρ(ψ)
```

However, the function explicitly requires computing a set of messages by reading *backwards* because of the ancestor axis on the second line. Although the ancestor axis could be evaluated in a streaming fashion by carefully memorizing appropriate pieces of the source, at the moment we found no tool capable of doing so.

A second possibility is to define a recursive, user-defined function that computes the Boolean value of the operator, as follows:

```
declare function local:until(ρ) {
  if (ω_ρ(ψ)) then true else
  if (ω_ρ(φ)) then local:until(ρ/message)
  else false };
```

The translation of an LTL formula using this version of the "until" does not require any backward computation. The translation method becomes a little more involved: one new until_i function must be declared for every occurrence of **U** in the formula, since ψ and φ, which both translate as XQuery expressions, cannot be passed as arguments to an XQuery function and must therefore be translated directly into the body of the function. Yet again, we found no tool capable of handling *recursive* user-defined functions.

This does not mean that **U** does not belong to the forward-only fragment —indeed, we just showed that there exists a way to compute **U** without memorizing or backtracking. The problem is rather linked with current implementations of XQuery engines.

At the moment, the only workaround we can suggest is to support the "until" operator at the top level of a formula only. When the $\text{LTL}^{\rightarrow}$ formula to evaluate is of the form $\varphi\,\mathbf{U}\,\psi$, we monitor two formulæ separately: $\mathbf{G}\,\varphi$ on one side, and $\mathbf{F}\,\psi$ on the other. It suffices to keep the monitors running until either $\mathbf{F}\,\psi$ announces true (in which case $\varphi\,\mathbf{U}\,\psi$ is fulfilled), or $\mathbf{G}\,\varphi$ announces false (in which case $\varphi\,\mathbf{U}\,\psi$ is violated). This solution is a reasonable compromise: it allows the use of **U** and extends the range of useful properties that can be expressed with $\text{LTL}^{\rightarrow}$, but since this operation is done only if **U** is the top-level operator, the constant space requirement still holds.

4.4 Experimental Results

To assess the tractability of runtime monitoring using XML methods, we performed a series of experiments on partial XQuery engines. There have been ample discussions about benchmarking issues of various XQuery engines [1]; the purpose of our experiments is not to benchmark query engines, but rather to get an early empirical validation of the translation provided in Section 3 in choreography scenarios similar to the one presented in Section 2.1.

Table 1: Average and maximal processing time in milliseconds per message, for each of the four web service interaction specifications, and each of the four XQuery engines tested: (S)axon, (D)DXQ, (M)XQuery and (X)QPull.

	Spec. #			
	1	2	3	4
S	1.4 (8.6)	2.6 (16.1)	1.5 (8.8)	1.5 (11.3)
D	1.0 (5.6)	2.0 (11.8)	1.0 (5.7)	0.9 (4.3)
M	2.2 (12.7)	4.4 (18.0)	18.3 (32.7)	4.3 (28.7)
X	233 (2477)	455 (4934)	259 (2763)	1.0 (5.8)

To this end, we produced 100 trace files of length ranging from 10 to approximately 800 messages, each containing a randomly created sequence of the messages described in the above scenario. Each of these traces was then sent to two commercial XQuery engines with partial streaming support (Saxon and DataDirect XQuery) and two academic engines with full streaming support (XQPull and MXQuery). These engines evaluated the XQuery translation of Interaction Specifications 1–4 in streaming mode. We then computed the processing overhead required by measuring the average and maximum CPU elapsed time per message. A summary of the results is shown in Table 1.

These results show that for the two commercial tools, it took in average 1 to 2 milliseconds, and never more than 16 milliseconds, to process one message of a trace. This tends to show that monitoring web service choreographies through streaming XML engines can actually be done in real time.

As a fully streaming engine, MXQuery performed reasonably well on most properties, especially compared with commercial engines with partial streaming support. One notable exception is Specification 3: for 38 of the 100 trace files (those with more than 470 messages), the engine failed to complete the evaluation of the query as it ran out of the 128 MB of memory allowed to the Java virtual machine (these traces are not included in the computation of the average time). The three other tools we tested did not take more time or memory to evaluate Specification 3 compared to the others. The experimental XQPull performed more slowly than the other tools. It computed Specification 4 very quickly compared to the three other properties; however it returned results which differed from the other three engines for some of the trace files; therefore, the very small validation times reported cannot be trusted.

These results show that, for the moment, fully streaming engines are more experimental and fragile in nature than commercial-grade products with limited streaming support. The forward-only fragment of LTL can therefore be seen as a "safe zone" where more stable tools can be used to perform runtime monitoring.

5. CONCLUSION

We have shown in this paper how web service choreography specifications formalized in Linear Temporal Logic can be monitored at runtime using XQuery engines with capabilities for streaming XML. To our knowledge, this is the

first application of streaming XML features to perform runtime monitoring. This approach has the advantage of taking profit of existing machinery already available in web service execution environments.

However, the processing of XQuery expressions on streaming XML is still an open problem, and many engines provide only little support for streaming. The forward-only fragment of LTL was studied in response to this observation; it represents the "least-common denominator" that XQuery engines with limited streaming capabilities can still support. Experimental tests on existing, off-the-shelf commercial XQuery tools showed that runtime monitoring could be added with really minimal modifications to existing execution environments, provided that they include an XQuery engine with minimal streaming support. These encouraging results open the way to the development of more refined XML-based methods to perform runtime monitoring, and to a more precise characterization of the forward-only fragment of LTL.

6. REFERENCES

[1] L. Afanasiev and M. Marx. An analysis of XQuery benchmarks. *Inf. Syst.*, 33(2):155–181, 2008.

[2] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model (DOM), W3C Recommendation, 1998. http://www.w3.org/TR/REC-DOM-Level-1/.

[3] 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 *Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, pages 87–107. Springer, 2003.

[4] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the memory requirements of XPath evaluation over XML streams. In *PODS*, pages 177–188. ACM, 2004..

[5] 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.

[6] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath processing with forward and backward axes. In *ICDE*, pages 455–466. IEEE Computer Society, 2003.

[7] I. Botan, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending XQuery with window functions. In *VLDB*, pages 75–86. ACM, 2007.

[8] D. Cacciagrano, F. Corradini, R. Culmone, and L. Vito. Dynamic constraint-based invocation of web services. In *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 138–147. Springer, 2006.

[9] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *ICSE*, pages 221–230. IEEE Computer Society, 2004.

[10] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. In *ICDE*, page 79. IEEE Computer Society, 2006.

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

[12] G. Decker, J. M. Zaha, and M. Dumas. Execution semantics for service choreographies. In *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2006.

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

[14] L. Fegaras, R. K. Dash, and Y. Wang. A fully pipelined XQuery processor. In *XIME-P*, 2006.

[15] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. Software Eng.*, 31(12):1042–1055, 2005.

[16] Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O'Farrell, and J. Waterhouse. Runtime monitoring of web service conversations. In *CASCON*, pages 42–57. ACM, 2007.

[17] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106. Morgan Kaufmann, 2002.

[18] 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.

[19] S. Hallé and R. Villemaire. XML methods for validation of temporal properties on message traces with data. In *CoopIS/DOA/ODBASE*, volume 5331 of *Lecture Notes in Computer Science*, pages 337–353. Springer, 2008.

[20] 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.

[21] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An optimizing XQuery processor for streaming XML data. In *VLDB*, pages 1309–1312. Morgan Kaufmann, 2004.

[22] X. Li and G. Agrawal. Efficient evaluation of XQuery over streaming data. In *VLDB*, pages 265–276, ACM, 2005.

[23] S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield. SOAP service description language (SSDL). Technical Report CS-TR-899, University of Newcastle, Newcastle upon Tyne, 2005.

[24] J. H. Park and J.-H. Kang. Optimization of XQuery queries including for clauses. In *ICIW*, page 37. IEEE Computer Society, 2007.

[25] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD Conference*, pages 431–442. ACM, 2003.

[26] M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming xquery evaluation. In *ICDE*, pages 236–245. IEEE, 2007.

[27] M. F. J. Siméon, C. Chen, B. Choi, V. Gapeyev, A. Marian, P. Michiels, N. Onose, D. Petkanics, C. Rath, C. Ré, M. Stark, G. Sur, A. Vyas, and P. Wadler. Galax, an XQuery implementation. http://www.galaxquery.org.

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