Runtime Enforcement of Web Service Message Contracts with Data

Sylvain Hallé[†], *Member, IEEE,* Roger Villemaire[‡], *Affiliate Member, IEEE*

Abstract—An increasing number of popular SOAP web services exhibit a *stateful* behaviour, where a successful interaction is determined as much by the correct format of messages as by the sequence in which they are exchanged with a client. The set of such constraints forms a "message contract" that needs to be enforced on both sides of the transaction; it often includes constraints referring to actual data elements inside messages. We present an algorithm for the runtime monitoring of such message contracts with data parameterization. Their properties are expressed in LTL-FO$^+$, an extension of Linear Temporal Logic that allows first-order quantification over the data inside a trace of XML messages. An implementation of this algorithm can transparently enforce an LTL-FO$^+$ specification using a small and invisible Java applet. Violations of the specification are reported on-the-fly and prevent erroneous or out-of-sequence XML messages from being exchanged. Experiments on commercial web services from Amazon.com and Google indicate that LTL-FO$^+$ is an appropriate language for expressing their message contracts, and that its processing overhead on sample traces is acceptable both for client-side and server-side enforcement architectures.

Index Terms—Web services, runtime monitoring, temporal logic

1 Introduction

From a messaging point of view, web service interactions can be considered as processes where the input and output of operations is composed of self-contained units ("messages") formed of various data elements. The precise format in which such operations can be invoked is detailed in an interface specification; for SOAP-based interactions, the Web Service Description Language (WSDL) [1] provides a way of defining the structure and acceptable values for XML requests and responses exchanged with a given service. A third-party, such as a web application developer, is required to comply with this specification to ensure a successful interaction with the service. In such a framework, each request-response pattern is supposed *stateless* and independent of any history.

Yet, we shall see in Section 2 that two popular web services, the Amazon e-Commerce Service and the Google Checkout service, rather exhibit stateful behaviour, where a successful interaction requires messages to be exchanged according to additional, *sequential* constraints. Furthermore, many "data-aware" properties are such that the the sequence of messages and their content are interdependent. Due to the intended stateless nature of web services, existing interface specification languages such as WSDL do not allow such dependencies to be specified. Most of them can be found by perusing the plain-text documentation of a service. Consequently, web service validation approaches focusing on WSDL compliance lack the ability to enforce complex sequential patterns of interaction at runtime.

In this paper, we present an algorithm for the runtime enforcement of such data-aware web service *message contracts*. To express them using a uniform formal notation, in Section 3 we introduce LTL-FO$^+$, an extension of the well-known Linear Temporal Logic (LTL) providing first-order quantification over the data inside a trace of XML messages. We show how LTL-FO$^+$ is suitable for expressing the message contracts for Amazon's and Google's examples.

In Section 4, we develop a 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 precompute an automaton, to store previous messages or to keep in memory anything except its current symbolic 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. We developed an open source, freely available Java applet called BeepBeep, that can be used for runtime enforcement both on JavaScript clients and Java-based servers. In Section 5, the results of these experiments are presented and discussed. They indicate that runtime monitoring of LTL-FO$^+$ can be performed in real-world scenarios and does not impose a large processing overhead, even for transactions of 10,000 messages and data domains of up to 1,000 elements.

The contributions of this paper are manifold. First, the paper provides an original study of web service interface specifications: it extracts and formalizes a number of constraints where sequential and data constraints are intermixed. It provides a simple and efficient algorithm for the runtime enforcement of these constraints and demonstrates its soundness. To the best of our knowledge, this is also the first systematic empirical study of a runtime enforcement

[†]*Université du Québec à Chicoutimi, Canada; e-mail: shalle@acm.org.*
[‡]*Université du Québec à Montréal, Canada; e-mail: villemaire.roger@-uqam.ca.*

approach on commercial web services, with large message sequences and realistic data domains.

2 Constraints on Web Service Message Sequences

The analysis of web service messaging interfaces can follow a "behaviourist" conception similar to the way it is actually invoked by a third party: one knows of a web service only what he can observe by interacting with it. This includes any message sent to the service, and any response emitted by that service. The internal state of a service, the contents of any database it can query are therefore irrelevant to our study. In this perspective, a *valid* interaction with a service is simply a sequence of messages (received and sent) which does not contain errors, and whose responses correspond to the behaviour from that service.

In the following, we present two real-world web services, taken from major players in the field. We show that for both of them, valid, error-free interactions consist not only of individual valid messages, but that values in various messages of a *sequence* are actually interdependent.

2.1 The Amazon E-Commerce Service

Recent statistics indicate that more than 330,000 developers are registered in Amazon's programs and that its daily web service transactions consume more bandwidth than the Amazon.com web site itself [2]. Among the broad palette of offerings, a straightforward one is the Amazon E-Commerce Service (ECS) [3], which makes Amazon.com's inventory available through a web service interface. In addition to simple search and browsing functionalities, the ECS also provides shopping cart manipulation operations that allow a client to create an order.

The semantics of the ECS operations follows the natural understanding of how a shopping cart should be handled: operations such as ItemSearch, CartCreate, CartAdd, CartRemove, CartModify and CartClear are self-explanatory. As expected, error messages will be sent if nonsensical sequences of commands are attempted. A simple example is the following:

Runtime Property 1. *Until a cart is created, the only operation allowed is ItemSearch.*

As the ECS documentation specifies, trying to perform any operation on a cart requires a cart ID, which is only returned as the response to a CartCreate message. Therefore, until that time, the only remaining valid operation is ItemSearch; performing any other operation will result in a AWS.ECommerceService.InvalidCartId or AWS.MissingParameters error messages. Similarly, cart manipulation follows a couple of natural rules:

Runtime Property 2. *A client cannot remove something from a cart that has just been emptied.*

As the ECS documentation specifies, doing so will result in the AWS.InvalidParameterValue error message being sent to the application.

However, while it is reasonable to believe that any quality application will actually check this obvious property by itself, the ECS documentation lists other constraints which are much less "natural"; an example is the following:

Runtime Property 3. *A client cannot add the same item twice to the shopping cart.*

This constraint, which is indeed part of the documentation, has nothing to do with the "natural" properties of a shopping cart, but rather deals with the particular implementation of the ECS. The service expects any modifications to items already present in the cart to be made through an Edit message; adding one more item therefore becomes *editing* the quantity of that item (in this case, incrementing it by one). Adding an existing item through the Add operation is greeted by the AWS.ECommerceService.ItemAlreadyInCart error message.

The previous constraints deal with requirements on the service consumer side which, if violated, provoke error messages by the ECS. Conversely, some constraints can also be elicited from the service's responses to a client's requests. For example:

Runtime Property 4. *A shopping cart created with an item should contain that item until it is deleted.*

Constraints on message sequences and parameters are not specific to shopping cart-like transactions. Requirements of a similar nature can be found in other Amazon web services, such as the Amazon Fulfilment Web Service [4].

2.2 Google Checkout

The sequential nature of these constraints is not the result of an exceptional design decision specific to Amazon. To prove our point, we provide an additional study from a second major web service provider, Google. A commercial web service suite provided by the Google company, called the Google Checkout API, allows to transfer money to and from credit card and bank accounts between its registered members or other financial institutions [5]. In addition to direct usage by individuals, an organization wishing to use these functionalities from its own web site can do so through Google's web service API.

This service involves a number of interactions between a client and the server. In particular, the server periodically sends notifications about the status of a pending order. The following constraint, taken from the Express Checkout documentation, indicates that the consumer of the service is not allowed to continue a transaction until a special notification has been received:

Runtime Property 5. *Before shipping the items in an order, the client should wait until it has also received the risk information notification for that order as well as the order state change notification informing the client that the order's financial state has been updated to* CHARGEABLE.

Moreover, some constraints in Google's documentation

also relate to the timing between different messages sent and received:

Runtime Property 6. *In a notification-history-request message, the start-time element's value must not be within 30 minutes of the time that the API request is submitted, or more than 450 days earlier than the time that the API request is submitted.*

The key point in these examples is that assuming "reasonable" interaction with a service is not enough to prevent error-free communications. Implementation details, specific to a particular instance of a service, create additional constraints on the possible operations that cannot be guessed unless explicitly specified in some way. Moreover, these two examples are not exceptional. Other web service contexts where the sequence of messages must be taken into account have been described [6]–[9].

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; to repair an invalid transaction or to determine the actual cause of the violation, our runtime enforcement mechanism can be used to call arbitrary functions defined by a developer.

2.3 A Case for Runtime Enforcement

As these examples show, the combination of sequential and data constraints forms a *message contract* that interface description languages such as WSDL are not designed to express. Nevertheless, both the service provider and an eventual third-party client must abide with these constraints to ensure a successful interaction: failure to do so results in non-sensical interactions, and in most cases, to error messages.

One possible way to ensure compliance to these constraints is to formalize them, and monitor them at runtime. Such a mechanism can be implemented and used in various ways.

In **client-side enforcement**, a client interacting with some web service is given the formal contract definition and applies it to the sequence of messages it sends and receives. The runtime enforcement mechanism acts as a "safety net" that prevents the relay of non compliant messages to the web service in case the client's application logic is faulty. The same monitor can be used to ascertain that the web service itself fulfils its part of the contract, and returns what the contract says it should.

In **server-side enforcement**, an external runtime monitoring module intercepts the messages at the interface on the web service's side and performs the same checks as for client-side monitoring. Depending on the specification to enforce, a monitor can even be used to detect malicious patterns of non compliant behaviour such as cross-site scripting or replay attacks.

The monitoring task can also be sent to an external, trusted third-party called a **protocol controller** [10]. Internal checks can be bypassed if both sides trust that this controller will block any non-compliant sequences of messages. Finally, a runtime monitor can be used in offline mode on a pre-recorded trace of messages to perform **log analysis**.

Several arguments in favour of runtime enforcement approaches have been put forward [6]. 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 a process was initially 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 [11]. For example, on the theoretical level, web services communicate through channels of potentially infinite length, thereby rendering the general model checking problem undecidable unless 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. [12] 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: [13] 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. [14] 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.

3 Formalizing Message Contracts

Sequential properties of interactions are generally expressed with a variant of a state machine or temporal logic. However, as will be detailed in Section 6, none of the representations we surveyed were appropriate in our context. Many missed the quantification over the data fields of the messages required in our examples. Those that did imposed restrictions on quantification that made them unsuitable for a web service scenario.

The logic we present in this section is LTL-FO$^+$, a first-order extension of a well-known logic called Linear

Temporal Logic (LTL); LTL has already been suggested for the static verification of web service interface contracts [15]–[17]. Although we concentrate on LTL, our approach is general: many other languages can be mapped into equivalent LTL expressions, or extensions thereof; this includes, among others, Message Sequence Charts [18], SSDL's Message Exchange Patterns (MEP) and Rules protocol frameworks [19], and the *Let's Dance* choreography description language [20].

3.1 Messages and Traces

LTL has been introduced to express properties about sequences of states in systems called Kripke structures [21]. In the present case, the states to be considered are messages inside a conversation. Formally, let us denote by M the set of XML messages. A sequence of messages $m_1, m_2 \ldots$, where $m_i \in M$ for every $i \geq 1$, is called a message trace. We write m_i to denote the i-th message of the trace $\overline{m}$, and $\overline{m}^i$ to denote the trace obtained from $\overline{m}$ by starting at the i-th message.

A domain function is used to fetch and compare values inside a message; it receives an argument π representing a *path* from the root to some element of the message. This path is defined using standard, XPath 1.0 notation. Formally, if we let D be a domain of values, and Π be the set of XPath expressions, the domain function Dom is an application $M \times \Pi \rightarrow 2^D$ which, given a message $m \in M$ and a path $\pi \in \Pi$, returns a subset $Dom_m(\pi)$ of D, representing the set of values appearing in message m at the end of the path π. For example, if we let Π be the set of XPath formulæ, $\pi \in \Pi$ be the particular formula "/message/stock/name", and $m \in M$ be the following message:

```
<message>
  <action>placeBuyOrder</action>
  <stock>
    <name>stock-1</name>
    <amount>123</amount>
  </stock>
  <stock>
    <name>stock-2</name>
    <amount>456</amount>
  </stock>
</message>
```

then $Dom_m(\pi) = \{\text{stock-1}, \text{stock-2}\}$.

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

LTL-FO$^+$'s syntax is based on classical propositional logic, using the connectives $\neg$ ("not"), $\vee$ ("or"), $\wedge$ ("and"), $\rightarrow$ ("implies"), to which four temporal operators have been added. An LTL-FO$^+$ formula is a well-formed combination of these operators and connectives, according to the usual construction rules:

Definition 1 (Syntax)**.** 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 $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\mathbf{G}\,\varphi$, $\mathbf{F}\,\varphi$, $\mathbf{X}\,\varphi$, $\varphi\,\mathbf{U}\,\psi$, $\varphi\,\mathbf{V}\,\psi$ 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.*

The semantics of each of these symbols is then defined as follows:

$\overline{m} \models c_1 = c_2$	$\Leftrightarrow$	c_1 is equal to c_2
$\overline{m} \models \neg\varphi$	$\Leftrightarrow$	$\overline{m} \not\models \varphi$
$\overline{m} \models \varphi \vee \psi$	$\Leftrightarrow$	$\overline{m} \models \varphi$ or $\overline{m} \models \psi$
$\overline{m} \models \mathbf{F}\,\varphi$	$\Leftrightarrow$	$\overline{m}^i \models \varphi$ for some $i \geq 1$
$\overline{m} \models \mathbf{X}\,\varphi$	$\Leftrightarrow$	$\overline{m}^2 \models \varphi$
$\overline{m} \models \varphi\,\mathbf{U}\,\psi$	$\Leftrightarrow$	$\overline{m}^j \models \psi$ for some j and $\overline{m}^i \models \varphi$ for $i < j$
$\overline{m} \models \exists_\pi x_i : \varphi$	$\Leftrightarrow$	$\overline{m} \models \varphi[b/x_i]$ for some $b \in Dom_{\overline{m}_1}(\pi)$

TABLE 1
Semantics for LTL-FO$^+$

Definition 2 (Semantics)**.** *We say a message trace $\overline{m}$* satisfies *the LTL-FO$^+$ formula φ, and write $\overline{m} \models \varphi$ if and only if it respects the rules in Table 1. As usual, we define the semantics of the other connectors with the following identities: $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$, $\mathbf{G}\,\varphi \equiv \neg(\mathbf{F}\,\neg\varphi)$, $\varphi\,\mathbf{V}\,\psi \equiv \neg(\neg\varphi\,\mathbf{U}\,\neg\psi)$, $\forall_p x : \varphi \equiv \neg(\exists_p x : \neg\varphi)$.*

Boolean connectives carry their usual meaning. The temporal operator **G** means "globally": the formula $\mathbf{G}\,\varphi$ means that formula φ is true in every message of the trace. The operator **F** means "eventually"; the formula $\mathbf{F}\,\varphi$ is true if φ holds for some future message of the trace. The operator **X** means "next"; it is true whenever φ holds in the next message of the trace. Finally, the **U** operator means "until"; the formula $\varphi\,\mathbf{U}\,\psi$ is true if φ holds for all messages until some message satisfies ψ.

3.3 LTL-FO$^+$ for Message Contracts

Equipped with this semantics, we can revisit the previous examples and show how runtime properties can be expressed as LTL-FO$^+$ formulæ.

The simplest of them is Runtime Property 2, which stipulates that a CartRemove message cannot be sent for a cart that has just been cleared (and still does not contain any item). Its formalization in LTL-FO$^+$ is straightforward:

$$\mathbf{G}\,(\forall_{\text{CartClear/CartID}}\, c_1 : ((\forall_{\text{CartRemove/CartID}}\, c_2 : c_1 \neq c_2)\ \mathbf{W}\,(\exists_{\text{CartAdd/CartID}}\, c_3 : c_1 = c_3)))$$

The **W** operator is called "weak until"; the formula $\varphi\,\mathbf{W}\,\psi$ is similar to $\varphi\,\mathbf{U}\,\psi$, except that ψ is not required to eventually hold as long as φ remains true.[1] The formula says that globally, every time a cart ID i_1 is seen in a CartClear message, then this cart ID is not seen in

1. Formally: $\varphi\,\mathbf{W}\,\psi \equiv (\varphi\,\mathbf{U}\,\psi) \vee (\mathbf{G}\,\varphi)$.

a CartRemove message until it appears in a CartAdd message. This indeed ensures that one cannot attempt to remove something from a cart that has just been cleared.

The previous formula only invokes cart IDs at multiple moments in the trace. Runtime Property 3 requires to combine cart and item IDs, as follows:

$$\begin{aligned}&\mathbf{G}\,(\forall_{\text{CartCreate/Items/Item/ASIN}}\, i_1 :\\ &\quad \mathbf{X}\,(\forall_{\text{CartCreateResponse/CartID}}\, c_1 :\\ &\quad \mathbf{G}\,(\forall_{\text{CartAdd/CartID}}\, c_2 : ((c_1 = c_2) \rightarrow\\ &\quad (\forall_{\text{CartAdd/Items/Item/ASIN}}\, i_2 : i_1 \neq i_2)))))\end{aligned}$$

This formula says that globally, for every item i_1 found in a CartCreate request and every cart ID c_1 found in its subsequent response, the following holds: from now on, every CartAdd message involving a cart ID c_2 is such that, if c_2 is the same as c_1, then none of the item IDs in that CartAdd message is equal to i_1. Hence, any item used to create a cart cannot be later added to that same cart, which is equivalent to Runtime Property 3.

For the sake of completion, we provide the LTL-FO$^+$ translation of Runtime Property 4, which is:

$$\begin{aligned}&\mathbf{G}\,(\forall_{\text{CartAdd/Items/Item/ASIN}}\, i_1 : (\forall_{\text{CartAdd/CartID}}\, c_1 :\\ &\quad \varphi(c_1, i_1)\,\mathbf{W}\,\psi(c_1, i_1))))\end{aligned}$$

The formulæ $\varphi(c_1, i_1)$ and $\psi(c_1, i_1)$ take c_1 and i_1 as parameters and are defined as:

$$\begin{aligned}\varphi(c_1, i_1) = \forall_{\text{CartGetResponse/CartID}}\, c_5 : (c_1 = c_5 \rightarrow\\ \exists_{\text{CartGetResponse/Items/Item/ASIN}}\, i_5 : i_1 = i_5)\end{aligned}$$

$$\begin{aligned}\psi(c_1, i_1) = \forall_{\text{CartRemove/CartID}}\, c_4 :\\ (c_1 = c_4 \wedge \exists_{\text{CartRemove/Items/Item/ASIN}}\, i_4 : i_1 = i_4)\end{aligned}$$

Informally, $\varphi(c_1, i_1)$ expresses the fact that the current message is a CartGetResponse with a cart ID equal to c_1 and which contains an item ID equal to i_1; $\psi(c_1, i_1)$ expresses the same fact, but about a CartRemove message. Runtime Property 4 then says that for every item i_1 found in a CartAdd for cart ID c_1, then i_1 appears in every CartGetResponse for that cart until it is removed from the cart. At this point, the reader should be convinced that the runtime properties shown in Section 2 can be expressed into equivalent LTL-FO$^+$ formulæ. We omit the translation of the remaining runtime properties.

The amount of effort required to formulate LTL-FO$^+$ message contracts is not as daunting as it appears. First of all, extracting the plain-text constraints from the English documentation of a service should not count as part of the task, since these constraints must always be taken into account by the developer when writing an application, even if they are not to be formally monitored. The intent to monitor merely provides an incentive to collect them in a systematic way.

Once each constraint is identified, it generally provides all the necessary information to formalize it into LTL-FO$^+$. This amounts to writing the path expressions that fetch relevant attributes for each message, and inserting LTL operators in between to specify the proper temporal relations. Writing down the expressions for the examples in this paper were a matter of a few minutes: they contain at most five or six path expressions, and two to three temporal operators.

3.4 Extension to 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 [22] demonstrate. The Google Express Check-out service shows examples of such constraints; Runtime Property 6 refers to time, and time differences between messages.

The present framework handles it by adding a timestamp τ to each message. The actual timestamp need not even be exchanged through messages, but quantification on τ simply amounts to fetching the current timestamp from the system's clock. In the same way as [23], metric temporal logic then becomes a particular case of data parameterization. For example, Runtime Property 6 becomes:

$$\begin{aligned}&\mathbf{G}\,(\forall_{\text{action}}\, a : \forall_{\text{start-time}}\, t :\\ &\quad a = \text{“notification-history-request”} \rightarrow\\ &\quad (|t - \tau| > 30 \wedge |t - \tau| < 450 \times 1,440))\end{aligned}$$

It specifies that the `start-time` element of any notification-history-request message must be at last 30 minutes, and at most 450 days away from τ, the current value of the system clock.

4 Runtime Enforcement of Message Contracts

Work on runtime monitoring begins with a classical result on Linear Temporal Logic:

Theorem 1 (From e.g. [24])**.** *For every LTL formula φ, there exists a Büchi automaton M such that for every infinite trace σ, we have that $\sigma \models \varphi$ if and only if $\sigma \in L(M)$.*

In other words, this result indicates that given an LTL formula φ, there exists a Büchi automaton that accepts exactly the infinite traces satisfying φ. Performing LTL runtime monitoring becomes straightforward: it suffices to build this automaton M, determinize it if required, and then "follow" a path in M as a particular trace is read. The trace violates φ if, at some point, no valid transition exists from the current state of M, given the next event to read. This is the approach followed in e.g. [25].

4.1 Monitoring with Data

A problem arises with this approach when quantification on data elements is introduced, as is the case with LTL-FO$^+$. If the domains are infinite or not known in advance, the Büchi automaton becomes impossible to build. Even if the domains are finite, they can be large enough to prevent

automaton construction as well (think of the case where a property is expressed on an item ID, which can be anything in Amazon's catalogue).

A key observation comes from the fact that in runtime monitoring, the Büchi automaton can be constructed *on-the-fly*. Only the start state of the automaton is initially "built". Then, as the trace is progressively read, only the parts of the automaton relevant to that particular trace can be expanded. This result is interesting for classical LTL in itself: a propositional version of this algorithm is used internally by the SPIN model checker to transform an LTL formula into a Büchi automaton [26]. However, it becomes crucial in LTL-FO$^+$ runtime monitoring. Instead of generating all states for all possible values of a given parameter, an on-the-fly algorithm will only create the states corresponding to values that have actually been *observed*. Since all messages are finite (yet unbounded), for any finite prefix of any trace, the set of observed values will be finite, and so will be the number of states of the partial Büchi automaton.

This in turn is only possible because of the particular definition of message quantification in LTL-FO$^+$, where a quantifier applies to values fetched in the *current* message. Hence the LTL-FO$^+$ formula $\forall_\pi x : \mathbf{F}\,\varphi(x)$ indicates that the values at the end of path π in the *first* message of the trace (i.e. $Dom_{m_1}(\pi)$) all eventually reappear to fulfil φ. Using a classical first-order quantifier, the previous formula would rather mean that all values x in some domain $Dom.(\pi)$ (independent of any message) eventually satisfy $\varphi(x)$. This is impossible to verify if the domain is not known in advance, or if it is infinite.

4.2 On-the-Fly Monitoring Algorithm 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 3 (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;*
- *$\delta : Q \times M \to Q$ 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 q'. 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" [27], or implemented as an external observer acting as a verifying layer between acting parties [13].

```
δ(q, m)
  q' = ∅
  For each N = Γ ⊩ Δ ∈ q
    N' = Δ ⊩ ∅
    q' = q' ∪ UPDATE(m, N')
  End for
  Return q'
End function
```

TABLE 2
The function δ changes the state of the watcher based on a message observed in the trace.

Although LTL-FO$^+$ is similar in many respects to another logic called CTL-FO$^+$, the model checking algorithm developed in [8] 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.

The algorithm is inspired from [26] 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 the identities in Definition 2 and the formulæ $c_1 \neq c_2 \equiv \neg(c_1 = c_2)$ and $\mathbf{X}\,\varphi \equiv \neg(\mathbf{X}\,\neg\varphi)$.

4.2.1 Transition Function

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 2.

The update function δ 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 UPDATE on that resulting node. UPDATE takes a node and decomposes the formulæ from its left-hand side according to the rules shown in Table 3. 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.

This form of decomposition is like a variant of sequent calculus [28] applied to temporal formulæ. Intuitively, the function UPDATE decomposes and evaluates all the LTL-FO$^+$ formulæ that must be true in the current state, eventually evaluating quantified variables and replacing equalities with their Boolean value. At the same time, UPDATE 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 $\overline{m} = \overline{m}_1\overline{m}_2 \dots \overline{m}_n$, we define $\delta(q, \overline{m}) = \delta(\delta(\dots\delta(\delta(q, \overline{m}_1), \overline{m}_2)\dots), \overline{m}_n)$.

It shall be noted that the monitoring algorithm does not require any *a priori* knowledge of the message's structure or its elements' types. Message contents are only relevant when evaluating a first-order quantifier, and this only applies on-the-fly to the current message, by fetching any values occurring at the end of a path. If the path does not exist in the message, the quantifier simply evaluates on the empty set of values.

4.2.2 Acceptance Conditions

It remains to determine how the watcher can conclude that a trace fulfils or violates a property. To this end, a set of three outcomes is used: $\top$ indicates that the property is fulfilled, $\bot$ indicates that the property is violated, and "?" indicates an inconclusive result: the property cannot be guaranteed to be neither true, nor false. The outcome function assigns to each possible watcher state one of these outcomes.

Definition 4 (Outcome function)**.** *Let q be a watcher state, and $O = \{\top, \bot, ?\}$ be the set of outcomes. The outcome function f is defined as follows:*

$$f(q) = \begin{cases} \top & \text{if } \emptyset \Vdash \emptyset \in q \\ \bot & \text{if } q = \emptyset \\ ? & \text{otherwise} \end{cases}$$

The violation condition is straightforward: if a call to δ 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.

Finally, when neither result can be concluded from the current state of the watcher, the outcome function returns the "inconclusive" result.

4.3 Correctness and Complexity

To prove the soundness and completeness of the runtime monitoring algorithm, we must first assert the correctness of the structural decomposition defined in Table 3.

Theorem 2. *For a set of k monitor nodes $\{N_1, \dots, N_k\}$ of the form $N_i = \Gamma_i \Vdash \Delta_i$ ($0 \leq i \leq k$), define*

$$\Psi(\{N_1, \dots, N_k\}) = \bigvee_{i=1}^{k} \left(\left(\bigwedge \Gamma_i \right) \wedge \mathbf{X} \left(\bigwedge \Delta_i \right) \right) \quad (1)$$

where $\bigvee \emptyset = \mathit{false}$ and $\bigwedge \emptyset = \mathit{true}$ (therefore, $\Psi(\emptyset) = \mathit{false}$ and $\Psi(\{N_1, \dots, N_k\}) = \mathit{true}$ as soon as $N_i = \emptyset \Vdash \emptyset$ for some N_i). Let $N = \Gamma \Vdash \Delta$ be a node and $N_i = \Gamma_i \Vdash \Delta_i$ ($0 \leq i \leq k$) be the k nodes resulting from the application of one of the rules in Table 3. Let $\overline{m} = m, m_1, m_2, \dots$ be an arbitrary trace of messages. Then $\overline{m} \models \Psi(\{N\}) \Leftrightarrow \overline{m} \models \Psi(\{N_1, \dots N_k\})$.

Proof: The proof is done by showing how the equivalence is preserved by the application of each transformation rule. We omit the trivial cases of operators $\neg$, $\vee$ and $\wedge$, as well as temporal operators **X**, **G**, **F**, **U** and **W**, where the respective parts of the proof in [26] can be adapted. We still need to prove ground equality and quantification, as follows.

1) $N = c_1 = c_2, \Gamma \Vdash \Delta$. Two cases must be considered. First, if $m \models c_1 = c_2$, then applying the decomposition rule results in a single node $N_1 = \Gamma \Vdash \Delta$. Hence:

$$\begin{aligned} \overline{m} \models \Psi(\{N\}) &\Leftrightarrow \overline{m} \models c_1 = c_2 \wedge \Gamma \wedge \mathbf{X}\,\Delta \\ &\Leftrightarrow m \models c_1 = c_2 \text{ and } \overline{m} \models \Gamma \wedge \mathbf{X}\,\Delta \\ &\Leftrightarrow \overline{m} \models \Gamma \wedge \mathbf{X}\,\Delta \\ &\Leftrightarrow \overline{m} \models \Psi(\{N_1\}) \end{aligned}$$

   Second, if $m \not\models c_1 = c_2$, then applying the decomposition rule does not produce any node and returns an empty set. The case for $c_1 \neq c_2$ is symmetrical.

2) $N = \forall_p x : \varphi, \Gamma \Vdash \Delta$. Applying the decomposition for the universal quantification yields a single node of the form $N_1 = \varphi[x/b_1], \dots, \varphi[x/b_n], \Gamma \Vdash \Delta$, where $\{b_1, \dots, b_n\} = Dom_m(p)$. Then:

$$\begin{aligned} \overline{m} \models \Psi(\{N\}) &\Leftrightarrow \overline{m} \models \forall_p x : \varphi \wedge \Gamma \wedge \mathbf{X}\,\Delta \\ &\Leftrightarrow \overline{m} \models \varphi[x/b_1] \wedge \dots \wedge \varphi[x/b_n] \\ &\qquad \wedge \Gamma \wedge \mathbf{X}\,\Delta \\ &\Leftrightarrow \overline{m} \models \Psi(\{N_1\}) \end{aligned}$$

3) $N = \exists_p x : \varphi, \Gamma \Vdash \Delta$. Applying the decomposition for the existential quantification spawns n

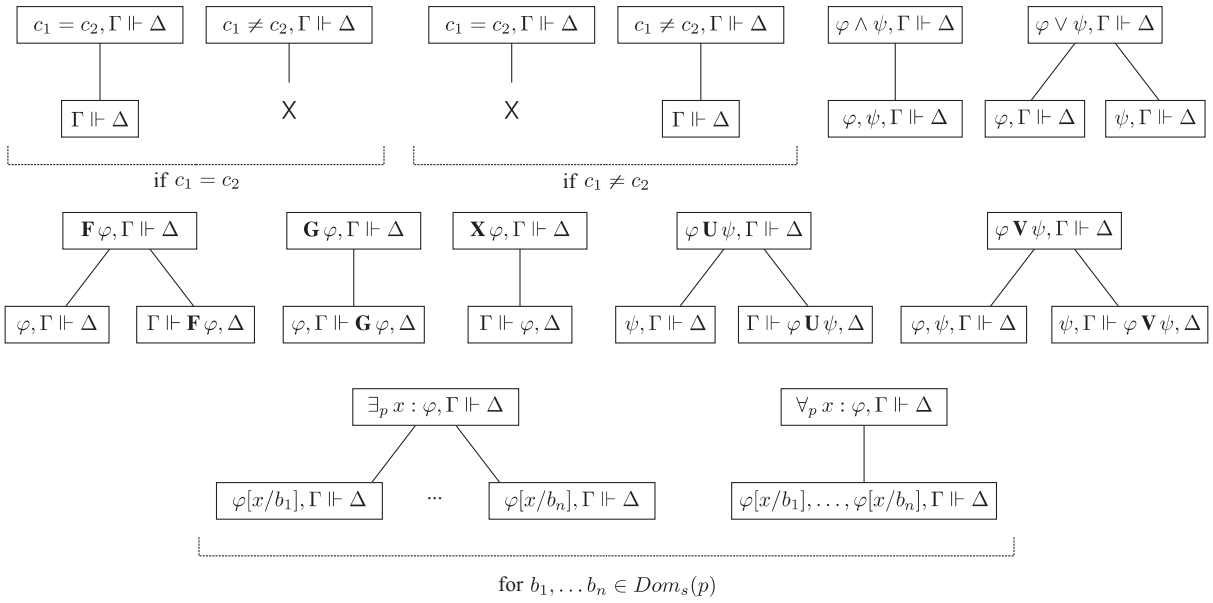

TABLE 3
Decomposition rules for a watcher's state node.

nodes of the form $N_i = \varphi[x/b_i], \Gamma \Vdash \Delta$, where $\{b_1, \ldots, b_n\} = Dom_m(p)$. Then:

$$\begin{aligned}
\overline{m} \models \Psi(\{N\}) &\Leftrightarrow \exists_p x : \varphi \wedge \Gamma \wedge \mathbf{X}\Delta \\
&\Leftrightarrow \overline{m} \models (\varphi[x/b_1] \vee \cdots \vee \varphi[x/b_n]) \\
&\qquad \wedge \Gamma \wedge \mathbf{X}\Delta \\
&\Leftrightarrow \overline{m} \models (\varphi[x/b_1] \wedge \Gamma \wedge \mathbf{X}\Delta) \vee \ldots \\
&\qquad \vee (\varphi[x/b_n] \wedge \Gamma \wedge \mathbf{X}\Delta) \\
&\Leftrightarrow \overline{m} \models \Psi(\{N_1\}) \vee \cdots \vee \Psi(\{N_n\}) \\
&\Leftrightarrow \overline{m} \models \Psi(\{N_1, \ldots, N_n\})
\end{aligned}$$

□

One can observe that for a message m and a node $\Gamma \Vdash \emptyset$, the result of UPDATE(m, N) is a set of nodes $\{N_1, \ldots, N_n\}$, such that for every node N_i, we have $N_i = \emptyset \Vdash \Delta_i$. Indeed, the repeated application of the rules in Table 3 progressively decomposes the formulæ in Γ until only atoms of the form $c_1 = c_2$ or $c_1 \neq c_2$ remain. Each atom is then evaluated, which results either in its deletion from the set Γ, or in the deletion of the node altogether. The function UPDATE simply repeats this process for every node in a state q. This observation allows us to say:

Theorem 3. *Let $\overline{m} = m_1, m_2, m_3, \ldots$ be a trace of messages, and $q = \{N_1, \ldots, N_n\}$ be the set of nodes in a watcher such that every N_i is of the form $N_i = \emptyset \Vdash \Delta_i$. Let $q' = \delta(q, m_2)$. We have that $m_1, m_2, m_3, \ldots \models \Psi(q)$ if and only if $m_2, m_3, \ldots \models \Psi(q')$.*

Proof: By definition:

$$\begin{aligned}
m_1, m_2, m_3, \ldots \models \Psi(q) &\Leftrightarrow m_1, m_2, m_3, \ldots \models \mathbf{X}\Gamma \\
&\Leftrightarrow m_2, m_3, \ldots \models \Gamma \\
&\Leftrightarrow m_2, m_3, \ldots \models \Psi(\Gamma \Vdash \emptyset)
\end{aligned}$$

Then, by Theorem 2, this is equivalent to $m_2, m_3, \ldots \models \Psi(\delta(\Gamma \Vdash \emptyset, m_2))$. □

It remains to show that the outcome function correctly assigns a truth value to the state of a watcher.

Theorem 4. *Let φ be an LTL-FO$^+$ formula, $\overline{m} = m_1, m_2, \ldots, m_k$ be a finite trace of messages, and $q = \{N_1, \ldots, N_n\}$ be the set of nodes in a watcher such that $q = \delta(\emptyset \Vdash \varphi, \overline{m})$. If $f(q) = \bot$ (resp. $f(q) = \top$) then for any infinite continuation $\overline{m}'$ of $\overline{m}$, $\overline{m}, \overline{m}' \not\models \varphi$ (resp. $\overline{m}, \overline{m}' \models \varphi$).*

Proof: By Definition 4, $f(q) = \bot$ implies that $q = \emptyset$. By repeated application of Theorem 3, we have:

$$\begin{aligned}
\overline{m}, \overline{m}' \models \varphi &\Leftrightarrow \overline{m}, \overline{m}' \models \Psi(\emptyset \Vdash \varphi) \\
&\Leftrightarrow \overline{m}' \models \Psi(q) \\
&\Leftrightarrow \overline{m}' \models \Psi(\emptyset)
\end{aligned}$$

But $\Psi(\emptyset) = false$ by Theorem 2, and hence $\overline{m}, \overline{m}' \not\models \varphi$. The proof for the case $f(q) = \top$ is similar. □

This result entails that, when the outcome function returns ? after reading some finite prefix of a trace, this prefix can be extended by at least one more message without provoking a direct violation of the specification. Note that this does not mean that the trace itself is guaranteed to be compliant; for example, the property $\mathbf{X}\bot$ cannot be satisfied by any infinite trace, yet one needs to reach the second message of that trace to explicitly violate it. In line with this observation, the outcome function returns ? until the second state is read.

Although complexity cannot be demonstrated due to lack of space, an intuitive argument places it on par with classical LTL. In the worst case, a Büchi automaton is exponential in the size of the LTL formula it is based on.

Since LTL-FO$^+$ subsumes LTL, the same result applies. This exponential upper bound, however, is mitigated by the fact that the algorithm works on-the-fly, and only generates at any step a small subset of the complete automaton.

4.4 Further Refinements

A number of natural refinements can be applied to this basic algorithm.

4.4.1 Anticipative Semantics

The previous observation shows that the outcome function looks one message ahead when computing its truth value. Therefore, it does not identify a violation that will manifest itself later on, but is unavoidable nonetheless. One might therefore be interested in an outcome function with a greater lookahead, that would return $\bot$ as soon as no valid continuation of a trace is possible, even though the current prefix does not explicitly violate the specification yet —this is what [29] calls an *anticipative semantics*.

While in the previous example, recognizing this fact is easy, one can devise other cases for which this identification is less trivial. For example, the formula $\mathbf{G}\,\neg\varphi \wedge \mathbf{F}\,\varphi$ cannot be true on any infinite trace, since it asserts both that φ is never true, and that φ eventually will become true. Moreover, φ can itself be an arbitrarily complex temporal expression, and its two occurrences can even be written in equivalent, but completely different ways that makes hard to realize **G** and **F** actually have the same argument.

In fact, an outcome function f' could be devised so that $f'(q) = \bot$ when $\Psi(q)$ is a contradiction, i.e. when $\Psi(q)$ is unsatisfiable. However, deciding satisfiability of "plain" LTL is PSPACE-complete, so that satisfiability of LTL-FO$^+$ is PSPACE-hard. Moreover, such a computationally-intensive decision algorithm must be repeated at every new message, in order to check that no contradiction is generated by the last application of UPDATE.

In contrast, the function f defined in this paper is a simpler form of outcome function that recognizes $\Psi(\emptyset)$, but not other, more complex forms of contradictions such as the ones previously discussed. However, it runs in constant time; in addition, from our experimental study of real-world services in Section 5, we discovered that such anticipative semantics was not required.

4.4.2 Additional Truth Values

Since most observed traces will be finite, it is possible that the watcher comes to a state where neither the acceptance, nor the rejection condition applies, although no further message is coming. An example of this is the formula $\mathbf{G}\,\varphi$ for any formula φ. This formula can never become true on a finite prefix of any trace, as the next message can always violate φ. At this point, since φ is neither confirmed nor violated, the result is inconclusive. This readily entails that the watcher must return at least *three* values: "true", "false", and "?".

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 [29]; the discussion could be adapted to LTL-FO$^+$ as well.

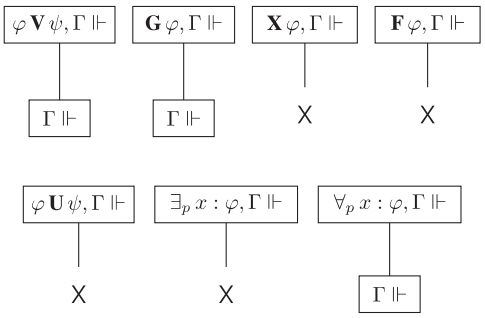

TABLE 4
Modified decomposition rules for an watcher's state node when the last message is reached.

Therefore, one can run UPDATE once more on the state of the watcher, but using a modified set of decomposition rules for UPDATE, shown in Table 4.

The rule for the temporal operator $\mathbf{G}\,\varphi$ can be interpreted as follows: the property φ must be true for all future messages of the trace; since no such message exists, the property is vacuously true and can therefore be eliminated from the node. 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 false, hence suppressing the node. The modified rules for the other temporal operators and quantifiers can be deduced in the same way.

Remark that these modified decomposition rules ensure that every branch will either terminate with an "X" or with the "accept" node $\emptyset \Vdash \emptyset$. Therefore, either the acceptance or the violation condition will apply to this last state and a partial conclusion can be drawn.

This conclusion differs from the regular acceptance and violation condition in that the result is temporary: an acceptance on the last message actually indicates that the property has not *yet* been violated, but could have been if further messages were sent. In the same way, a violation on the last message actually indicates that the property has not *yet* been fulfilled, but could be if further messages were sent. An outcome function f' can be defined, which returns these two additional outcomes at the end of a trace.

5 Experimental Results

Although the soundness and usefulness of this algorithm have been proved in theory, its practical value still needs

to be assessed on real-world scenarios. To this end, 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 The BeepBeep Runtime Monitor

We developed BeepBeep, a lightweight runtime monitor running as a Java applet.[2] This applet is responsible for actually keeping track and analyzing the incoming and outgoing messages with respect to an interface contract. It is a direct implementation of the runtime enforcement algorithm described in Section 4, and can accept as its input specification any LTL-FO$^+$ formula.

A monitor is instantiated by passing as an argument a character string containing a text representation of the LTL-FO$^+$ formula to monitor. This formula can be fetched for example, from a local contract file or a command line parameter. BeepBeep's implementation contains classes and data structures to convert and manipulate these formulæ internally. Once instantiated, the monitor's interface provides a method called `processMessage()`, which takes as an argument a String representation of an XML message and updates its internal state according to the specification. A second message, called `getOutcome()`, returns the result of the outcome function f on the current internal state.

A standard Ajax application communicates with a web service by sending and receiving SOAP messages through the standard XMLHttpRequest object provided by the local browser. BeepBeep can be used on the client side, inside these Ajax web applications. A JavaScript wrapper file overloads XMLHttpRequest with a class that behaves in the same way, with the exception that incoming and outgoing messages, before being actually sent (or returned), are deviated to the applet and possibly blocked if violations are found.

Including BeepBeep into an existing Ajax application is simple. It suffices to host two files (the `.jar` applet and the `.js` include) in the same directory as the Ajax application, and to load BeepBeep by adding a single line at the beginning of the original client's code. No other changes to the code are required: from this point, BeepBeep intercepts the messages and transparently monitors the conversation.

When BeepBeep detects that a message violates a contract property, its default behaviour is to block the message and to pop a window alerting the user, showing the plain-text description associated with that property. Alternatively, BeepBeep can be asked to call a function, called a *hook*, provided by the application developer. The developer can insert arbitrary code in that function, setup a breakpoint, and extract any debugging information deemed necessary. Therefore, BeepBeep can be used to form the basis of a powerful debugging tool for message contract violations.

2. BeepBeep and its source code are available for download under a free software license at http://beepbeep.sourceforge.net/.

Besides its ease of use, the main advantage of BeepBeep is that the specification of the contract is completely decoupled from the code required for its actual monitoring. The contract is located on the server side in a file separate from the monitor itself. This is in contrast with [27], [30], which require the compilation of a contract into executable Java code —an operation which must be repeated whenever the contract is changed. This requirement is ill-suited to the highly volatile nature of web service interactions. In BeepBeep, changing the contract can be done dynamically without changing anything to the clients: the algorithm is applied at runtime, and in the same way, to any LTL-FO$^+$ formula passed to the monitor.

The BeepBeep applet can also be wrapped for use as a server-side runtime enforcement tool, or as a protocol controller. Its use as a log analyzer, on pre-recorded traces of messages, is also straightforward.

BeepBeep monitors conversations specified at the XML message level; it is independent from any client implementation and does not refer to any internal variable of the client's source code. It is therefore non-invasive and can enforce specifications transparently with minor changes to the code, apart from including BeepBeep. Other approaches, such as [27], require heavier code instrumentation in order to correctly intercept non-compliant behaviour.

BeepBeep also has a low footprint: the total volume that needs to be downloaded by an Ajax application using BeepBeep (JavaScript + applet) is less than 50 kB, and this must be done only once when the application starts. By today's standards, this is negligible. Although most related work on runtime enforcement do not provide easy access to implementations for comparison, one might consider this: loading the Google Maps page from an empty cache requires downloading a volume of about 400 kB of data, and typing "Montreal" in its location bar immediately triggers the download of another 400 kB. We therefore argue that, for standard Ajax applications, the addition of BeepBeep as a runtime monitor represents a marginal increase in the volume of downloaded data.

BeepBeep was designed for client-side monitoring, where a trace is a sequence of messages starting when a user connects to the site until the user checkouts or closes the browser. However, it can also work on the server side. Since the server is the endpoint of multiple parallel sessions with different clients, the monitor must distinguish between these sessions to properly enforce the message contracts. However, this filtering must be done by the server code anyway, either by passing a session ID as an XML element in the messages, or as a cookie in the HTTP request that carries this message. In the first case, it suffices to prefix each runtime property with a universal quantifier over session IDs that BeepBeep can observe like any other XML element. In the second case, it suffices for the server to associate one instance of BeepBeep for each thread created when a new session opens; each instance is hence only dispatched the messages specific to that session, before passing them on to the thread proper. In either case, the problem of checking interleaved client

	Operators	Quantifiers	Length
Runtime Property 1	1	2	17
Runtime Property 2	2	3	18
Runtime Property 3	2	4	22
Runtime Property 4	2	6	28

TABLE 5
Number of temporal operators, data quantifiers, and total number of symbols in each of Runtime Properties 1-4.

sessions reverts to having a series of watchers, each of which monitoring and evaluating one single trace. Neither requires any modification to the monitor itself.

5.2 Tests on Synthetic Traces

Using the Amazon ECS as an example, we generated random traces of request-response messages. Each trace contains a random number of cart creations, item searches, and cart add, edit, remove, and clear operations. These operations manipulate items from a pool of 1,000 possible IDs and a maximum of 10 simultaneous instances of shopping carts. We believe that these values greatly exceed the parameters of a typical shopping session driven by a single user. All these traces correspond to semantically *valid* interactions, i.e. the contents of each created shopping cart is tracked throughout the whole sequence, and only valid operations are allowed to be selected on each shopping cart at any time (for example, an *edit* operation is never attempted on an empty cart, etc.).

100 traces of lengths ranging from 10 to 10,000 messages were randomly produced. Each of these traces was then assigned to an instance of the BeepBeep runtime monitor, which evaluated Runtime Properties 1-4 on them. The experiments were ran on an Intel 2.67 GHz CPU under Windows XP, with an out-of-the box Java runtime environment.

5.2.1 Validation time

For each run, we first measured the average processing time per message, and plotted the results in Figure 1. Graphs (a) to (d) represent a progressive increase in the complexity of the property, both in the number of temporal operators and data quantifiers, as is summarized in Table 5: subfigure (a), corresponding to Runtime Property 1, is a formula with 1 temporal operator and 2 data quantifiers; subfigure (d) is a more complex formula, with 2 temporal operators and 6 data quantifiers.

This experiment had three purposes: first, to determine the typical order of magnitude required for processing a message; second, to determine if more complex formulæ result in longer processing times; third, to assess whether the monitor's performance degrades as a trace unfolds. Indeed, all the runtime properties described in Section 2 require the monitor to capture and preserve data values from past messages for comparison with future messages. An accumulation of historical data to seek through at every message could progressively slow down the monitor.

By observing the plots in Figure 1, one can see that validation time is similar for all four properties; at the scale of only a couple temporal operators and quantifiers, formula complexity hence has a negligible impact. The higher processing time per message in small traces can be explained by a fixed overhead of about 10 ms incurred when initializing the monitor at startup. This overhead is amortized in traces with more messages. Globally, one can safely conclude that the time required to process a message remains well under 1 millisecond, once the monitor is started. All but a dozen traces required more than 2 milliseconds per message to be monitored, for any of the four properties. Moreover, the plots clearly indicate that the presence of long traces does not adversely impact on validation time.

5.2.2 Monitor size

The previous findings show that the monitor's average processing time per message is reasonable, and is not affected by the presence of long traces. This is the case even if the properties to monitor require that data values from the past be saved for later comparison. However, it may well be possible that the accumulation of historical data, even if it does not adversely impact on processing time, still reaches unreasonable bounds in terms of memory required. A second experiment recorded the maximal size that the monitor's state reached while evaluating each trace. Its purpose was to determine whether that size grew with trace length, and to what extent this growth can be bounded.

We recall from Section 4.2 that the state of the monitor consists of a set of nodes, each of which contains a finite list of LTL-FO$^+$ subformulæ. The total number of subformulæ in all nodes of a state is hence a measure of its size, which in turn is directly proportional to the amount of memory used by BeepBeep to store it. The results are plotted in Figure 2.

The stepwise behaviour of Figures (a) and (b) can be explained by the properties they relate to. Runtime Property 1 stipulates that ItemSearch is the only allowed operation before CartCreate. The monitor can hence be in only two states. If CartCreate has not yet appeared, the monitor's state contains a single node of the form $\varphi \Vdash \emptyset$, where φ is Runtime Property 1, a formula of size 17. Once CartCreate has appeared once, the monitor does not require any further work and simply propagates the empty node $\emptyset \Vdash \emptyset$, of size 0. Hence, the monitor's maximum state size reached in any trace is either 0 (if CartCreate is the first message of the trace), or 17 (otherwise). This is irrespective of the length of the trace, and is consistent with the plot in Figure 2(a).

Intuitively, the monitor's state for Runtime Property 2 grows each time a cart c is cleared, and includes a new instance of the LTL-FO$^+$ subformula stipulating that no CartClear must appear with c until a CartAdd operation involves c. This subformula is removed from the state as soon as CartAdd is called for cart c. Hence the monitor's state size evolves in discrete steps, with its maximum being a direct function of the number of carts that are simultaneously empty in a trace. Since the traces contained

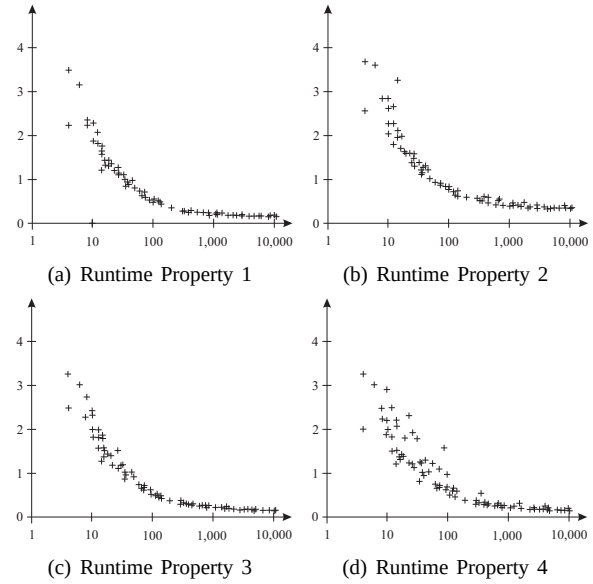

Fig. 1. Validation time per message (in milliseconds) according to trace length, for Runtime Properties 1-4.

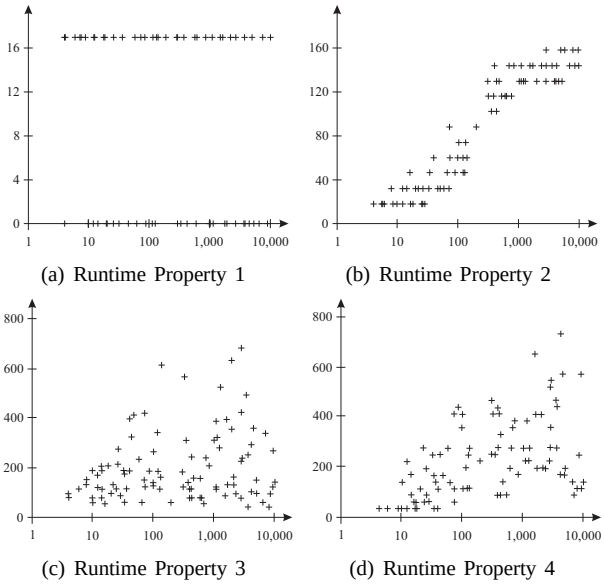

Fig. 2. Maximum memory consumption, in number of subformulæ, according to trace length, for Runtime Properties 1-4.

at most 10 carts, there are 11 possible values for the number of empty carts, and the plot contains 11 steps.

A similar reasoning could be applied to explain the patterns found in Figures (c) and (d). The bottom line of this analysis is that trace length is seldom the contributing factor that predicts memory consumption. For Runtime Property 1, the maximum size was determined by the presence of CartCreate at the first position in the trace; for Runtime Property 2, memory consumption is proportional to the number of empty shopping carts at any one time. In many cases, however, longer traces increase the probability that this contributing factor appears more often, and hence results in larger state sizes.

Nevertheless, the number of subformulæ that need to be stored by the watcher, and hence the memory footprint of the algorithm, remains within reasonable bounds and grows very slowly (i.e. logarithmically) with respect to trace length. Although the machine used for the experiments provided 8 GB of RAM, only a tiny fraction of it is actually required for the monitoring: for all but one trace, the peak number of ground terms in the monitor's state remained under 1,000.

5.3 Tests with Actual Web Services

The previous experiments allowed us to measure a number of factors impacting the monitor's performance in a controlled way. However, the absolute overhead of 1 ms per message should also be related to the processing time per message of an actual web service. To this end, BeepBeep has been tested on real Ajax applications in various scenarios.

We compared a plain Ajax client using a real-world web service, the Amazon E-Commerce web service, against the same client communicating through BeepBeep and monitoring 11 different contract properties. Since we did not have access to Amazon's file server, the contract file was located on the same server as BeepBeep for the needs of the experiment. Each version of the client sent to Amazon the same set of randomly generated message sequences; the difference in the elapsed time was measured. Since the experiment involved actual communications with the service, it was repeated on 20 different traces to average out punctual differences caused by the variable latency of the network.

Our findings indicate that on a low-end computer (Asus EeePC with a 600 MHz processor running Mozilla Firefox 2), monitoring LTL-FO$^+$ contract properties produces an average overhead of around 3%. As a rule, the state of the network accounts for wider variations on processing time than the additional computations required by the monitor. These results suggest that even on small devices such as smartphones and PDAs, the addition of a runtime monitor on the client side should not have noticeable effects on performance for typical web applications.

6 Discussion and Related Work

A number of approaches to the runtime monitoring of systems in general have been suggested over the years. However, the unique characteristics of web service message contracts makes each of them inappropriate in at least one respect, thereby warranting the development of LTL-FO$^+$. These characteristics are enumerated below.

6.1 Access to Data Parameters

Web service message contracts involve properties referencing data elements inside exchanged messages. Therefore, a first category of inappropriate monitoring solutions includes *propositional* runtime monitoring tools, where the sequence

of messages is analyzed, but the content of messages is abstracted away.

Enforcement monitors [31], a refinement of edit automata [32] and security automata [33], are special types of finite-state automata used to intercept, and possibly retain messages destined to a peer in a special waiting queue until one is guaranteed that sending these messages will not lead to a violation of the specification. In contrast, our monitoring approach discards messages that violate the specification. Since finite-state machines are used to represent the contracts, the messages are considered as atomic events with no data; this would not be suitable for the runtime properties we described in this paper.

Some of the runtime properties we presented include time constraints. Timed automata [34] can be used to represent such timing dependencies between events; they otherwise behave like classical automata and cannot quantify over data elements other than time; they hence cannot be used in the present context. Extended timed automata, used in model checkers like Uppaal [35], have data variables in addition to clocks; however, it is unclear how, for example, the variable (and potentially unbounded) list of item IDs in a shopping cart could be encoded with data variables in order to express Runtime Properties 3 and 4; a similar remark applies to the Input-Output State Transition Systems (IOSTS) used in [36] to specify monitoring properties. In addition, the overwhelming majority of properties of commercial web services elicit constraints on data elements such as shopping carts or item IDs, and seldom refer to time.

Similarly, propositional runtime monitoring was used by [12], 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. [27] 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*.

There also exist approaches that use logic, rather than finite-state machines, for specifying temporal constraints. An early work on this respect [37], which uses a rewriting approach similar to the one presented here to verify traces of events against LTL properties, using the Maude string rewriting engine. Since the logic used is classical LTL, the properties do not allow data parameterization. Similarly, different patterns of classical LTL properties were studied by [17], which introduces the concept of *obstacle* to detect possible violations of a specification. 2D-LTL [38] allows one to express correlations between parallel sessions occurring on a common timeline. However, apart from an implicit session identifier, no data is taken into account in the properties.

Finally, in [13] 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 to in the properties, but no quantification is allowed on data fields.

6.2 Complex Message Structure

If all possible parameter values are known in advance ($\{b_1, \ldots, b_n\}$), it is possible to create one propositional symbol for each value, and use propositional temporal logic to express a constraint. It suffices to repeat a formula $\varphi(x)$ for each possible value of x, e.g. $\varphi(b_1) \wedge \cdots \wedge \varphi(b_n)$. However, in the present context, the possible values are not known in advance: for example, Runtime Property 3, expressing a constraint on item IDs, would require to be repeated once for each possible ID in Amazon's catalogue, which is unrealistic. Therefore, the use of a quantified expression, $\forall x : \varphi(x)$, not only shortens the property, but also covers all possible values of x without knowing them in advance.

There exist a couple of approaches to runtime monitoring which allow some form of quantification on data fields. For example, usage control enforcement [39] uses plain first-order logic to access to data parameters inside events of the X11 windowing system. However, the language only expresses relations between one state and the next; complex temporal modalities, such as LTL's **F** or **U**, cannot be represented in the framework. [23] describes an algorithm for rule-based runtime monitoring, where the rules are temporal fixpoint functions that can include data arguments. [40] makes a similar use of data parameterization for a quantified variant of LTL.

However, in the current scenario, most messages not only contain an action name and a set of data parameters, but these parameters themselves are subject to a potentially complex XML structure. In the Amazon scenario, one cannot simply refer to "the" item ID in a shopping cart, as there can be multiple instances of the `ItemId` element in a message. A property can require that all, or only one of these item IDs fulfils a constraint, hence a form of quantification over message contents, including *existential* quantification, is required. This is probably the single most distinguishing point with respect to other verification applications. The aforementioned solutions work in a context where there is at most one instance of a parameter in a message, thus removing the need for quantification. This also rules out logics like $\mathcal{L}^*_{\mathcal{MDG}}$ [41] or Eagle [42], for the same reason. In contrast, BeepBeep can handle arbitrary nested structures; no upper bound on the arity of the messages needs to be fixed in advance.

6.3 Unrestricted use of quantifiers and temporal operators

A third characteristic of web service message contracts is that temporal operators and quantification must be mixed without limitation, as is the case in LTL-FO$^+$. In contrast, monodic temporal logic [43] requires that each subformula beginning with a temporal operator must have at most one

free variable. Apart from Runtime Property 2, none of the examples in our paper follow that restriction. Similarly, in an other extension of LTL called LTL-FO [44], quantifiers cannot be applied to formulæ containing temporal operators, except by taking the universal closure of the entire formula (which is not the case in our examples).

More closely related, [6] suggests a framework in which correlations between data in multiple messages are expressed and can be checked at runtime. However, to the best of our knowledge, the correlations imply a single request-response and do not involve messages arbitrarily far apart in time.

An alternative to state machines and temporal logics is the use of event calculus (EC), as is done in [7]. 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 [11] suggest. In this respect, we have shown 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 [45]. 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 toward runtime monitoring, but rather on static analysis.

Finally, a different approach has been proposed with specifications using XQuery on traces (SXQT) [46], 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. Such an adaptation is suggested in [47], where the use of *streaming* XQuery processors can read a trace progressively and output intermediate results, thereby working as a runtime monitor.

7 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 first-order quantification over message contents. As our study of commercial web services, such as Amazon ECS and Google Checkout, showed, the online documentation specifies a fair number of "data-aware" properties, where the sequence of messages sent to the service and the actual data values inside the messages define valid interactions.

An on-the-fly runtime monitoring algorithm was presented to enforce such constraints on message sequences. In addition to a proof of its soundness, this algorithm was implemented into a small Java applet, called BeepBeep, that can monitor and enforce any temporal property specified in LTL-FO$^+$. In particular, this applet can be used on the client side inside an Ajax web application and check all incoming and outgoing messages transparently for violations of the contract. Empirical studies on both Google and Amazon web services, for traces of 10,000 messages and data domains of 1,000 elements, show that the runtime enforcement of LTL-FO$^+$ can be practically and efficiently done with minimal modifications and negligible computing overhead, either on the client or server side.

The positive results obtained in this research project open the way to multiple extensions and improvements. Mappings between LTL-FO$^+$ and patterns in graphical languages such as UML could be devised, in order to ease the specification of runtime constraints on web services. The ultimate question remains as to who should provide these contracts and where they should be obtained. To further the reach of the tool to practitioners, the automated extraction of constraints from a service's source code or sample execution traces could entice service providers to systematically document them in some machine-readable form.

On a more technical side, the on-the-fly translation of LTL into an automaton presented here is just one of the many exiting ways of converting temporal logic into a finite-state equivalent; there exist many others [48], [49] that could also be extended with support for first-order quantification, and be experimented in their own right.

All in all, our study showed that the runtime enforcement of complex temporal specifications involving messages with data can be effectively done in real-world scenarios at a very low cost. By providing a transparent and very simple way of enforcing rich interface contracts into virtually any web service or Ajax client, BeepBeep contributes to increase the reach of logic and formal verification approaches in the development of everyday web applications.

Acknowledgements

The authors gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada and the Fonds québécois de recherche sur la nature et les technologies.

References

[1] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (WSDL) 1.1, W3C note," 2001. [Online]. Available: http://www.w3.org/TR/wsdl

[2] E. Schonfeld, "Amazon earnings call details: Web services use up more bandwidth than Amazon.com; the Kindle is a hit," 2008. [Online]. Available: http://tcrn.ch/caKitG

[3] "Amazon e-commerce service," 2005. [Online]. Available: http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl

[4] "Amazon fulfillment web service," 2005. [Online]. Available: http://docs.amazonwebservices.com/AWSFWS/1.0/DeveloperGuide/

[5] "Google checkout APIs," 2009. [Online]. Available: http://code.google.com/apis/checkout/

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

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

[8] S. Hallé, R. Villemaire, and O. Cherkaoui, "Specifying and validating data-aware temporal web service properties," *IEEE Trans. Software Eng.*, vol. 35, no. 5, pp. 669–683, 2009.

[9] S. Hallé, G. Hughes, T. Bultan, and M. Alkhalaf, "Generating interface grammars from WSDL for automated verification of web services," in *ICSOC-ServiceWave*, ser. Lecture Notes in Computer Science, L. Baresi, C.-H. Chi, and J. Suzuki, Eds., vol. 5900, 2009, pp. 516–530.

[10] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services, Concepts, Architectures and Applications*. Springer, 2004.

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

[12] 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*. New York, NY, USA: ACM, 2007, pp. 42–57.

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

[14] W. N. Robinson, "Monitoring web service requirements," in *RE*. IEEE Computer Society, 2003, pp. 65–74.

[15] S. Nakajima, "Lightweight formal analysis of web service flows," *Progress in Informatics*, no. 2, pp. 57–76, 2005.

[16] X. Fu, T. Bultan, and J. Su, "Analysis of interacting BPEL web services," in *WWW*, S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, Eds. ACM, 2004, pp. 621–630.

[17] W. Robinson, "A requirements monitoring framework for enterprise systems," *Requir. Eng.*, vol. 11, no. 1, pp. 17–41, 2006.

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

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

[20] G. Decker, J. M. Zaha, and M. Dumas, "Execution semantics for service choreographies," in *WS-FM*, ser. Lecture Notes in Computer Science, M. Bravetti, M. Núñez, and G. Zavattaro, Eds., vol. 4184. Springer, 2006, pp. 163–177.

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

[22] G. Governatori, Z. Milosevic, and S. W. Sadiq, "Compliance checking between business processes and business contracts," in *EDOC*. IEEE Computer Society, 2006, pp. 221–232.

[23] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *VMCAI*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds., vol. 2937. Springer, 2004, pp. 44–57.

[24] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Banff Higher Order Workshop*, ser. Lecture Notes in Computer Science, F. Moller and G. M. Birtwistle, Eds., vol. 1043. Springer, 1995, pp. 238–266.

[25] A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of real-time properties," in *FSTTCS*, ser. Lecture Notes in Computer Science, S. Arun-Kumar and N. Garg, Eds., vol. 4337. Springer, 2006, pp. 260–272.

[26] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *PSTV*, ser. IFIP Conference Proceedings, P. Dembinski and M. Sredniawa, Eds., vol. 38. Chapman & Hall, 1995, pp. 3–18.

[27] I. H. Krüger, M. Meisinger, and M. Menarini, "Runtime verification of interactions: From MSCs to aspects," in *RV*, ser. Lecture Notes in Computer Science, O. Sokolsky and S. Tasiran, Eds., vol. 4839. Springer, 2007, pp. 63–74.

[28] J. H. Gallier, *Logic for Computer Science: Foundation of Automatic Theorem Proving*. Longman Higher Education, 1986. [Online]. Available: http://www.cis.upenn.edu/~jean/gbooks/logic.html

[29] A. Bauer, M. Leucker, and C. Schallhart, "The good, the bad, and the ugly, but how ugly is ugly?" in *RV*, ser. Lecture Notes in Computer Science, O. Sokolsky and S. Tasiran, Eds., vol. 4839. Springer, 2007, pp. 126–138.

[30] G. Rosu, F. Chen, and T. Ball, "Synthesizing monitors for safety properties: This time with calls and returns," in *RV*, ser. Lecture Notes in Computer Science, M. Leucker, Ed., vol. 5289. Springer, 2008, pp. 51–68.

[31] Y. Falcone, J.-C. Fernandez, and L. Mounier, "Synthesizing enforcement monitors wrt. the safety-progress classification of properties," in *ICISS*, ser. Lecture Notes in Computer Science, R. Sekar and A. K. Pujari, Eds., vol. 5352, 2008, pp. 41–55.

[32] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: enforcement mechanisms for run-time security policies," *Int. J. Inf. Sec.*, vol. 4, no. 1-2, pp. 2–16, 2005.

[33] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.

[34] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, no. 126, pp. 183–235, 1994.

[35] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL - a tool suite for automatic verification of real-time systems," in *Hybrid Systems*, ser. Lecture Notes in Computer Science, R. Alur, T. A. Henzinger, and E. D. Sontag, Eds., vol. 1066. Springer, 1995, pp. 232–243.

[36] C. Constant, T. Jéron, H. Marchand, and V. Rusu, "Integrating formal verification and conformance testing for reactive systems," *IEEE Trans. Software Eng.*, vol. 33, no. 8, pp. 558–574, August 2007.

[37] K. Havelund and G. Rosu, "Testing linear temporal logic formulæ on finite execution traces," Tech. Rep., May 2001.

[38] F. Massacci and K. Naliuka, "Multi-session security monitoring for mobile code," Tech. Rep. DIT-06-067, November 2006.

[39] A. Pretschner, M. Büchler, M. Harvan, C. Schaefer, and T. Walter, "Usage control enforcement with data flow tracking for X11," in *Proc. 5th Intl. Workshop on Security and Trust Management (STM)*. Elsevier, 2009, pp. 124–137.

[40] V. Stolz, "Temporal assertions with parametrised propositions," in *RV*, ser. Lecture Notes in Computer Science, O. Sokolsky and S. Tasiran, Eds., vol. 4839. Springer, 2007, pp. 176–187.

[41] F. Wang, S. Tahar, and O. A. Mohamed, "First-order LTL model checking using mdgs," in *ATVA*, ser. Lecture Notes in Computer Science, F. Wang, Ed., vol. 3299. Springer, 2004, pp. 441–455.

[42] H. Barringer, D. Rydeheard, and K. Havelund, "Rule systems for run-time monitoring: From Eagle to RuleR," *Journal of Logic and Computation*, 2008.

[43] I. M. Hodkinson, "Complexity of monodic guarded fragments over linear and real time," *Ann. Pure Appl. Logic*, vol. 138, no. 1-3, pp. 94–125, 2006.

[44] A. Deutsch, L. Sui, V. Vianu, and D. Zhou, "Verification of communicating data-driven web services," in *PODS*, S. Vansummeren, Ed. ACM, 2006, pp. 90–99.

[45] C. E. Gerede and J. Su, "Specification and verification of artifact behaviors in business process models," in *ICSOC*, ser. Lecture Notes in Computer Science, B. J. Krämer, K.-J. Lin, and P. Narasimhan, Eds., vol. 4749. Springer, 2007, pp. 181–192.

[46] M. Venzke, "Specifications using XQuery expressions on traces," *Electr. Notes Theor. Comput. Sci.*, vol. 105, pp. 109–118, 2004.

[47] S. Hallé and R. Villemaire, "Runtime monitoring of web service choreographies using streaming XML," in *SAC*. ACM, 2009, pp. 1851–1858.

[48] C. Fritz, "Concepts of automata construction from LTL," in *LPAR*, ser. Lecture Notes in Computer Science, G. Sutcliffe and A. Voronkov, Eds., vol. 3835. Springer, 2005, pp. 728–742.

[49] P. Gastin and D. Oddoux, "Fast LTL to Büchi automata translation," in *CAV*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Springer, 2001, pp. 53–65.

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

Sylvain Hallé received the BS degree in mathematics from Université Laval in 2002 and the MSc in mathematics and PhD in computer science from Université du Québec à Montréal in 2004 and 2008, respectively. He was recently appointed as an assistant professor in the Department of Computer Science and Mathematics at Université du Québec à Chicoutimi, after completing a postdoctoral fellowship at at University of California, Santa Barbara. He received fellowships from the Natural Sciences and Engineering Research Council of Canada (NSERC) in 2005 and Quebec's Research Fund on Nature and Technologies (FQRNT) in 2008. His major research interests include Web applications and formal verification. He is a member of the ACM, the Association for Symbolic Logic, the IEEE, and the IEEE Computer Society. He was co-chair of DDBP 2008, TIME 2008 and the DDBP series of workshops from 2008 to 2010.

Roger Villemaire received the PhD degree from the University of Tübingen in 1988. He was a postdoctoral fellow at McGill University and later at Université du Québec à Montréal (UQAM). He is a professor in the Department of Computer Science at UQAM, which he joined in 1993. His research interests include applications of logic in computer science, in particular formalisms, methods and algorithms which can help to realize reliable computing systems. He was co-chair of TIME 2008 and served on its program committee in 2009. He is a member of the ACM, the Association for Symbolic Logic and the IEEE Computer Society.