Runtime Verification for the Web
A Tutorial Introduction to Interface Contracts in Web Applications

Sylvain Hallé[1] and Roger Villemaire[2]

[1] Université du Québec à Chicoutimi, Canada
shalle@acm.org
[2] Université du Québec à Montréal, Canada
villemaire.roger@uqam.ca

Abstract. This tutorial presents an introduction to the monitoring of web applications. These applications run in a user's web browser and exchange requests and responses with a server in the background to update their display. A demo application, called the Beep Store, illustrates why complex properties on this exchange must be verified at runtime. These properties can be formalized using an extension of Linear Temporal Logic called LTL-FO$^+$. The tutorial concludes with the presentation of BeepBeep, a lightweight runtime monitor for web applications.

1 Introduction

In the past decade, numerous applications, such as Facebook and Google Mail, have become part of popular culture. These so-called "web" applications come into the scope of a programming paradigm called *cloud computing*, where the user's web browser is responsible for loading from a server and displaying the various elements of the application's page. The user can interact with some of these elements, which in turn trigger the browser to send further requests to the server, and update the display.

To be properly understood by their respective recipients, each request and each response is expected to follow a specific structure, where the possible operations, parameters and values are precisely defined. In many cases, the browser-server exchange also moves forward according to a protocol, where the validity of a request depends on past events.

The technologies over which web applications are built were not designed with complex interactions in mind. Consequently, they do not provide facilities to define or enforce such an "interface contract". Ensuring a correct match between the browser's and the server's behaviour is an open problem, currently left as the developer's sole responsibility. Recording the sequence of requests and responses, and providing a means of preventing contract violations from occurring is an appealing prospect in this regard.

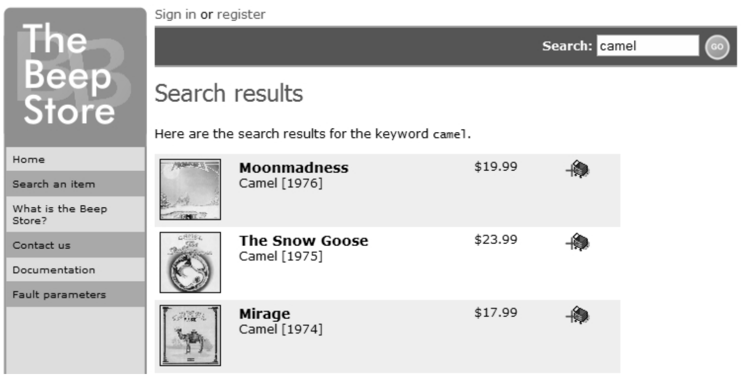
The present tutorial summarizes our experiments in the enforcement of interface contracts in web applications. Its interest lies primarily in providing a self-contained introduction to a domain that meets many favourable conditions for the application of runtime verification techniques. To this end, Section 2 presents a running web application typical of many real-world web services we studied in the past; Section 3 discusses the interface contract for this application. Section 4 introduces a formal language, LTL-FO$^+$, expressive enough for the constraints encountered, and describes how BeepBeep, a lightweight LTL-FO$^+$ runtime monitor, can be integrated into the initial application to effectively enforce the contract.

2 Anatomy of a Web Application: The Beep Store

For the purpose of this tutorial, we designed a simple web application called the Beep Store that will be used as a running example to illustrate web-based runtime verification concepts.

2.1 End-User Perspective

The Beep Store allows registered users to browse a fictional collection of books and music, and to manage a virtual shopping cart made of these elements. It runs out-of-the-box in any modern web browser pointed at the store's URL.[1]



Search results

Here are the search results for the keyword `camel`.

Moonmadness Camel [1976]	$19.99	
The Snow Goose Camel [1975]	$23.99	
Mirage Camel [1974]	$17.99	

Fig. 1. The Beep Store's web interface

Figure 1 shows a typical application screen. At any time, users can use the search box at the top right of the screen to type any keyword. Similarly, they can click on the "Search an item" menu element at the left to summon a more complete search pane, where they can restrict the search to a specific artist, a specific title, and split the result into pages of a fixed number of entries.

Pressing the "Go" button retrieves from the server the list of all relevant entries. Optionally, users have the option of adding an item from that list into

[1] `http://beepbeep.sourceforge.net/examples/beepstore`

a personal inventory called a "shopping cart". To do so, they must first log into the application (using the "Sign in" link at the top of the page) and provide their username and password. A shopping cart is automatically created when users add their first item into it.

Once a cart is created, a "Your Cart" button (not shown) appears at the right of the search box. Clicking this button opens the cart pane, which displays the list of all items currently in the user's cart, their quantity and total price. Buttons allow the user to edit the quantity for an item, or remove it altogether. Each action updates the cart's list on the fly.

Such a scenario is a purposefully condensed version of popular commercial web sites, such as Amazon or eBay. Indeed, although the Beep Store is a demo application, all its functionalities —and constraints on its use, as we shall see— have been found in at least one of the real-world web services we studied in the past [14]. This includes in particular the User-Controlled Lightpath Service [7], the Amazon e-Commerce Service [1], and the PayPal Express Checkout Service [2].

2.2 Internal Workings

Asynchronous JavaScript and XML (Ajax) refers to the collection of technologies used to develop such rich and interactive web applications. The execution of an Ajax application in a web browser is a straightforward process. First, the client's browser loads the application's page, `beepstore.html`. It uses it to render the page's content by interpreting its markup elements: text boxes, buttons, menu elements, headings, images. The header of this HTML file contains a link to a JavaScript document hosted in the same directory, called `beepstore.js`.

The JavaScript functions it contains are used for three purposes. First, it associates snippets of code to some page elements. For example, a button in the HTML file can be linked to a JavaScript function through the `onClick` event; any click on this button triggers the execution of the associated JavaScript function. Second, the web browser provides a JavaScript object, called `document`, whose methods can be used to access the HTML page's elements and modify their content and appearance dynamically. Hence, the button's `onClick` event can toggle the visibility of some page section that was previously hidden, producing an effect similar to a pop-up window. With proper coding, JavaScript can reproduce in the browser most of the look-and-feel of a traditional desktop application.

The last use of JavaScript is for the handling of requests and responses over the network. This is done through a standard object called `XMLHttpRequest`, also provided by the local browser.[2]

2.3 Interaction through XML

The second part of an Ajax application is a script running on the server side, and answering to requests initiated by the local browser's `XMLHttpRequest`

[2] An exception is Internet Explorer, which exposes the same functionalities under a different object called `MSXML`. Their differences are superficial.

object. In the case of the Beep Store, a PHP script called `beepstore.php` acts as the application's front door on the server. Data is exchanged using a standard markup called XML. Each XML document sent and received is called a *message*, and the communication between the browser and the server hence generates a message sequence.

Figure 2 shows the structure of two typical request-response pairs of messages sent by the Beep Store's application to its server. For instance, Figure 2(a) shows the message sent by the browser when a user clicks on the Login button: it includes an element called `Action` whose value indicates the name of the action to be executed by the server, and two additional parameters providing a `Username` and `Password`. The actual values inserted inside these two elements are dynamically fetched by the JavaScript function responsible for sending the Login message on the browser.

```
<Message>
 <Action>Login</Action>
 <Username>Sylvain</Username>
 <Password>banana</Password>
</Message>
```

(a) Login (request)

```
<Message>
 <Action>LoginResponse</Action>
 <SessionKey>123456</SessionKey>
</Message>
```

(b) Login (response)

```
<Message>
 <Action>CartCreate</Action>
 <SessionKey>123456</SessionKey>
 <Items>
  <Item>
   <ItemId>123</ItemId>
   <Quantity>1</Quantity>
  </Item>
  ...
 </Items>
</Message>
```

(c) Create a cart (request)

```
<Message>
<Action>CartCreateResponse</Action>
<SessionKey>123456</SessionKey>
<CartID>789123</CartId>
<Items>
 <Item>
  <ItemId>123</ItemId>
  <Quantity>1</Quantity>
  <Price>12.00</Price>
  <Author>The Beatniks</Author>
  <Title>Yelp!</Title>
 </Item>
 ...
</Items>
</Message>
```

(d) Create a cart (response)

Fig. 2. Examples of XML messages for the Beep Store

The server's PHP script processes this request by checking that the name-password pair is contained in its user database. In such a case, it creates and records a new unique session key, and produces the response message shown in Figure 2(b). The JavaScript code on the client side parses it and keeps the session key in local memory for future requests.

Request and response messages for cart creation, shown in Figure 2(c)-(d), are more complex. In addition to the `Action` and `SessionKey`, the creation request includes a compound element, `Items`, itself made of one or more `Item` elements. Each item specifies an item ID (taken from the store's catalogue) and the quantity of this item to be included in the cart. The response returned by the server repeats that information, provides a unique ID to the newly created cart, and adds pricing, title and author information for each item, as obtained from the store's database.

2.4 The Beep Store as a Web Service

One can see how the exchange of XML messages outsources the application's core functionalities to the server over the network, leaving the client with only the lighter, GUI-related processing. For example, database search and cart manipulations are handled by the server, which only sends the results of these operations to the browser for proper display. This architecture is appealing, if only for practical reasons: a browser-side search for an item would involve downloading the whole store's catalogue on the client.

As a matter of fact, the server's functionality is not limited to this particular web client: it is made publicly available as an instance of a *web service*. Any third-party developer can produce a working pair of HTML/JavaScript files and send requests to the Beep Store's PHP script; provided that the requests are properly formed and sent in a reasonable sequence, the store's script will serve them.

Similarly, a different server, accepting the same messages as the Beep Store, could be used indifferently by the web client. A web service can even send requests to another service. Ultimately, the vision of web services is to separate functionalities into simple, stand-alone units, communicating over the network through standardized mechanisms such as XML messaging. A web application is a particular case of this scenario consisting of a single browser-server pair.

3 Interface Contracts in Web Applications

The appealing modularity of web services is the source of one major issue: how can one ensure the interaction between each application and each service proceeds as was intended by their respective providers? Without any clear and mutual understanding of the acceptable requests and responses, an Ajax client might try to send a message that the server does not recognize, and vice versa. A correct interoperation between a client and a service is only guaranteed if both partners follow a well defined and enforceable *interface contract*.

3.1 The Beep Store Interface Contract

The source for such an interface contract invariably comes from the service's documentation, intended for developers. The online documentation for the Beep Store[3] is modelled after that of real-world web services, in particular the Amazon E-Commerce Service.

[3] `http://beepbeep.sourceforge.net/examples/beepstore/documentation`

The first observable part of an interface contract that this documentation provides consists of the description of all the XML request and response messages for each operation, in a way similar to Figure 2. Any client and service must produce messages following the structure mentioned there.

In addition, accompanying text explains the semantics of each operation, and lists a number of conditions that must be fulfilled for each operation to be properly processed and return a response. Some of these constraints have been purposefully integrated into the Beep Store to faithfully reproduce behaviour found in some real-world web service we studied. Our prior work led us to divide these constraints into three categories:

Data Constraints. The first class of properties expresses constraints over the structure and values inside a single message at a time. For example, in the `ItemSearch` message:

P1. The element `Page` must be an integer between 1 and 20.
P2. The element `Page` is mandatory only if `Results` is present; otherwise it is forbidden.

These requirements go beyond the specification of a rigid XML structure: they also provide ranges for possible values, and even state that the presence of some element be dependent on the presence of another. Further data constraints could, for example, impose possible values for some element as a function of the value in another element —an example of such a constraint can be found in the Amazon E-Commerce Service [12].

Control-Flow Constraints. Other restrictions are related to the sequence in which operations are invoked. Any application introducing the concept of session, or manipulating persistent objects such as a shopping cart, includes control-flow constraints of that kind. For example:

P3. The `Login` request cannot be resent if its response is successful.
P4. All cart operations, such as `CartCreate`, must follow a successful `Login-Response`.

These constraints introduce the notion of *state* into the application: the possible future messages allowed depend on what has happened in the past. Indeed, it does not make sense for a user to try to login again after a successful login. Similarly, since shopping carts must be associated to a logged user, it is impossible to create such a cart without first logging in. An attempt at such operations hints at some programming flaw on the client side, and should be replied by an error message from the server.

Data-Aware Constraints. Furthermore, the Beep Store includes properties referencing data elements inside exchanged messages, such that these data elements are taken at two different moments in the execution and need to be compared. Properties having this characteristic have been dubbed "data-aware" temporal properties [15]. For example:

P5. There can be at most one active cart ID per session key.
P6. You cannot add the same item twice to the shopping cart.

Property 5 obviously forbids a client to involve a `CartCreate` operation twice. However, it also requires that at any time, the `CartId` value found in a message be the same for all subsequent messages. This must be respected both by the client (which cannot try to sneak information about another cart by simply providing a different ID) and the server (which cannot change a cart's ID after it has been communicated to the client).

Property 6, although seemingly counter-intuitive, has actually been found in the Amazon E-Commerce Service, as reported in [14]. The service requires that, to add one more of an existing item into a cart, the `CartEdit` operation be invoked on that item instead of repeating a `CartAdd` message. Therefore, this property entails that any `ItemId` appearing in a `CartAdd` message no longer appears in a future `CartAdd` (unless the item is found in a `CartRemove` message in between).

The reader is referred to the Beep Store documentation for a list of all constraints in the interface contract; further examples of constraints in other scenarios can be found in our earlier papers [13, 15, 14].

3.2 Issues with Current Technologies

The examples shown above represent a small portion of all the constraints imposed by the Beep Store. The interface contract for a typical web service is made of dozens of such properties. However, as numerous and well-documented as these properties are, the technologies over which web applications are built bring a number of issues when it comes to handling them.

Free-Form Messages. As such, there is no "web service protocol". The closest one gets to such a concept is with the Simple Object Access Protocol (SOAP) [20], itself built as a special case of the HTTP protocol that web browsers have been using for decades. A SOAP request is little more than a collection of HTTP headers, followed by an XML payload formed of two mandatory sections: `Head` and `Body` (the XML documents in Figure 2 are sent inside the `Body`). Apart from these conditions, SOAP regards the payload as a free-form document. This entails that the message structure —the web equivalent of types in a classical programming language— is not even checked.

Stateful Behaviour, Stateless Protocol. HTTP is also a *stateless* protocol, where each new request processed by the server is detached from previous ones, and unrelated to those that follow. At the time HTTP was designed, this characteristic was appealing for its simplicity of implementation and the limited resources it requires for processing a request. Yet, we have seen how the Beep Store, typical of many web applications, requires long-running interactions spanning multiple requests and responses, and where past requests determine current valid ones.

Since session logic is not carried transparently through the protocol, it must be explicitly handled by the application itself. This is why the Beep Store must simulate sessions through a sequence of individual request-response pairs, where a unique identifier created at the start of a session (the `SessionKey`) is repeated in each subsequent message. The session's state (shopping cart contents, user name) is written to persistent storage between requests and can be retrieved using this identifier.

No Standardized Contract Notation. It follows from these observations that most properties of an interface contract lie at a higher conceptual level than current web protocols. Their expression and enforcement should therefore be handled in an extra control layer on top of HTTP and SOAP.

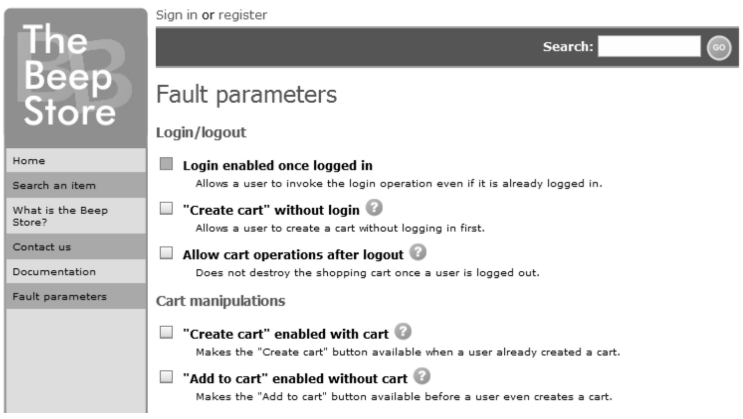
The only part of interface contracts that made it to some form of standardization is the Web Service Description Language (WSDL) [9]. WSDL allows the creation of an auxiliary document that specifies the XML structure of each request and response accepted by a service. Existing software frameworks, such as Apache Axis [3], can generate template functions called *stubs* for each message. By communicating only through these auto-generated stubs, a client or server can be guaranteed to send only WSDL-compliant messages. The same stubs can also verify at runtime that any incoming message follows the WSDL specification.

If the generation of WSDL-based stubs and the runtime verification of message structures is now considered routine, the Beep Store shows that there is much more to interface contracts than checking XML message structures: WSDL runtime verification only traps violations of Property 1. No standardized language exists to express Properties 2-6; no framework helps building an application that complies with them, or traps their violations at runtime. A developer needs to peruse the service's natural language documentation, and check each constraint manually with a copious amount of tests.

To illustrate this fact, the Beep Store browser client can be turned into a deliberately faulty application. Its user interface contains a "Fault Parameters" pane, shown in Figure 3, that provides the complete list of constraints specified by the store's documentation. Normally, the client is robust and performs thorough checks of all these constraints before sending any message to the server. For example, once a shopping cart is created, it hides the "Create cart" button to avoid users creating a second one (see P5). Similarly, it hides the Login button once a user has successfully logged in (see P4). With the Fault pane, the user can tick the checkbox for any of these constraints, causing the application to bypass these measures and allow actions at inappropriate moments.

3.3 Particularities of Web Service Interface Contracts

Web service interface contracts bear many resemblances with temporal properties or contracts found in other domains. In object-oriented languages, some classes, such as Java's `File` or `Iterator`, also impose constraints on the sequence of method calls; these class contracts can be checked at runtime using tools such



Fault parameters

Login/logout

■ **Login enabled once logged in**
Allows a user to invoke the login operation even if it is already logged in.

☐ **"Create cart" without login**
Allows a user to create a cart without logging in first.

☐ **Allow cart operations after logout**
Does not destroy the shopping cart once a user is logged out.

Cart manipulations

☐ **"Create cart" enabled with cart**
Makes the "Create cart" button available when a user already created a cart.

☐ **"Add to cart" enabled without cart**
Makes the "Add to cart" button available before a user even creates a cart.

Fig. 3. The Beep Store's "Fault Parameters" pane allows the application to deliberately ignore some elements of its interface contract, causing the server to reply with an error message on purpose

as Java-MOP [8]. Similarly, research on trace validation applied to spacecraft test sequences unveiled constraints that correlate both data values and ordering of events [6]. This hints that existing solutions developed for other scenarios could be ported to the web service realm. However, web services exhibits a combination of characteristics that makes them unique.

Data-Aware Dependencies. Simplified versions of the contract properties could be verified using classical Petri nets, finite state automata or propositional linear temporal logic. However, many constraints can only be faithfully checked by taking into account dependencies between data parameters. Obviously, the data elements cannot be enumerated statically: Property 6 would have to be repeated for every item in the Beep Store's catalogue, which would be required to be known in advance.

Data-aware dependencies do not merely require the access to parameters inside a message; they also need such values to be kept, and compared at a later time with values inside another message. Moreover, the time separating these two messages is unknown in advance, and potentially unbounded; hence it does not suffice to keep a fixed-size window of past messages.

Complex Message Structure. Not only do most messages contain an action name and a set of data parameters, these parameters themselves are subject to a potentially complex XML structure. In the Beep Store, 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 is required.

This is probably the single most distinguishing point with respect to other verification applications. Most verification solutions that take data dependencies into account work in a context where there is at most one instance of a parameter in a message (removing the need for quantification).

4 Runtime Verification of Interface Contracts

The previous sections described how the architecture of web applications, coupled with the state of current technologies, calls for a runtime verification solution of interface contracts. This section describes the authors' attempts at developing and running a possible solution. It first shows how the properties in Section 3 can be expressed in a formal language, called LTL-FO$^+$. It then presents BeepBeep, a Java-based runtime monitor for LTL-FO$^+$. BeepBeep can be integrated into the Beep Store described in Section 2, and enforce its interface contract at runtime.

4.1 Formalizing Contracts with LTL-FO$^+$

LTL-FO$^+$ is an extension of Linear Temporal Logic (LTL) developed to address the characteristics of web application interface contracts. Relating the expressiveness of this logic to other solutions has extensively been done in previous papers [15, 19].

Let Q be a set of queries, M a set of messages, and V a set of atomic values. A *query function* π is defined as $\pi : Q \times M \to 2^V$. Intuitively, $\pi(q, m)$ retrieves a set of values from a message m, given some "filtering criterion" q. We typically use as π the function that takes as query a path in an XML document (a slash-separated list of element names) and which returns all the values at the end of such a path in the current message. For example, in the following message m, we have $\pi($"`message/item`"$, m) = \{A, B\}$.

```
<Message>
  <Item>A</Item>
  <Item>B</Item>
  <Client>10</Client>
</Message>
```

A *message trace* is a sequence $\overline{m} = m_1 m_2 \dots$ such that $m_i \in M$ for $i \geq 1$; $\overline{m}^i$ denotes the suffix of $m_i m_{i+1} \dots$.

Definition 1 (Syntax). *The language LTL-FO$^+$ (Linear Temporal Logic with Full First-order Quantification) is obtained by closing LTL under the following construction rules:*

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

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 following rules: if φ is of the form $\neg\psi$, $\psi \vee \psi'$, $\mathbf{F}\,\psi$, $\mathbf{X}\,\psi$ and $\psi\,\mathbf{U}\,\psi'$, the semantics is identical to LTL's. Let $q \in Q$ be some query expression. The remaining cases are defined as:*

$$\overline{m} \models c_1 = c_2 \Leftrightarrow c_1 \textit{ is equal to } c_2$$
$$\overline{m} \models \exists_q x_i : \varphi \Leftrightarrow \overline{m} \models \varphi[b/x_i] \textit{ for some } b \in \pi(q, m_1)$$

We define the semantics of the other connectives with the usual identities: $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$, $\mathbf{G}\,\varphi \equiv \neg(\mathbf{F}\,\neg\varphi)$, $\varphi\,\mathbf{V}\,\psi \equiv \neg(\neg\varphi\,\mathbf{U}\,\neg\psi)$, $\forall_q x : \varphi \equiv \neg(\exists_q x : \neg\varphi)$.

Equipped with this language, it is possible to revisit the interface contract described earlier and formalize it with LTL-FO$^+$ formulæ. Properties 1 and 2 are data constraints; they only involve the temporal operator $\mathbf{G}$ to specify that the data constraint applies to all messages. If we define $q_1 =$ `Message/Action`, $q_2 =$ `Message/Page` and $q_3 =$ `Message/Results`, then Properties 1 and 2 become respectively equations 1 and 2 below:

$$\mathbf{G}\,(\forall_{q_1} a : a = \texttt{ItemSearch} \rightarrow (\forall_{q_2} p : p \geq 1 \wedge p \leq 20)) \tag{1}$$
$$\mathbf{G}\,(\forall_{q_1} a : a = \texttt{ItemSearch} \rightarrow (\exists_{q_3} r : \top \leftrightarrow \exists_{q_2} p : \top)) \tag{2}$$

The first property states that globally, if the message's action is `ItemSearch`, then for every `Page` value p inside that message, p is in the range $[1, 20]$. Similarly, the second property states that any `ItemSearch` message is such that for every `Results` element, a `Page` element must exist (π returns the empty set if no element with the specified path can be found in a message). The symbol $\top$ stands for "true"; $\exists_q x : \top$ is true whenever the path q exists.

In a similar way, control-flow properties P3 and P4 become formulæ 3 and 4 below:

$$\mathbf{G}\,(\forall_{q_1} a : a = \texttt{LoginResponse} \rightarrow (\mathbf{X}\,\mathbf{G}\,(\forall_{q_1} a' : a' \neq \texttt{LoginResponse})) \tag{3}$$
$$(\forall_{q_1} a : a \neq \texttt{CartCreate}\,\mathbf{W}\,(\forall_{q_1} a' : a' \neq \texttt{LoginResponse}) \tag{4}$$

Finally, by defining $q_4 =$ `Message/CartId`, $q_5 =$ `Message/SessionKey` and $q_6 =$ `Message/Items/Item`, data-aware properties 5 and 6 can be formalized into the following:

$$\mathbf{G}\,(\forall_{q_4} c : \forall_{q_5} k : \mathbf{G}\,(\forall_{q_4} c' : \forall_{q_5} k' : (k = k' \rightarrow c = c'))) \tag{5}$$

$$\begin{aligned}&\mathbf{G}\,(\forall_{q_1} a : a = \texttt{CartAdd} \rightarrow \\ &\qquad (\forall_{q_6} i : \mathbf{X}\,\mathbf{G}\,(\forall_{q_1} a' : a' = \texttt{CartAdd} \rightarrow \forall_{q_6} i' : i \neq i')))\end{aligned} \tag{6}$$

Equation 5 states that in every message, the presence of a `CartId` c and `SessionKey` k entails that, from that point on, any other occurrences of a `CartId` c' and `SessionKey` k' are such that the same key imposes the same ID. This is equivalent to P5. The "data-awareness" of this constraint can be observed in the fact that two variables that have been quantified across temporal operators (such as c and c') are compared at a later point in the expression.

A particularity of LTL-FO$^+$ lies in its quantification mechanism: note in the definition how the values over which quantification applies are only those found in the *current* message, m_1. For example, in equation 6, variables i and i' both quantify over catalogue item IDs. If quantification did not depend on the current message, the previous formula would always be false, as any value c bound to i would also be admissible for i', making the assertion $i \neq i'$ false at least once. The previous formula rather states that at any time in the execution of the application, for any item ID i appearing in a CartAdd message, then from now on in any *future* CartAdd message, any item ID i' is different from i. Hence, it will be true exactly when no item appears more than once in any CartAdd message, which is consistent with Property 6.

LTL-FO$^+$ allows the Beep Store to publicize a formal version of its interface contract. To this end, an auxiliary file, contract.txt, is hosted along with the Beep Store's other files on the server. It contains the list of all LTL-FO$^+$ formulæ forming that contract, including equations (1)-(6) described above. Figure 4 shows a snippet of the contract file containing a text rendition of Property 1.

```
% The page element must be an integer between 1 and 20
; G (([a /Message/Action] ((a) = ({ItemSearch}))) ->
    ([p /Message/Page] (((p) > ({1})) & ((p) < ({20})))))
```

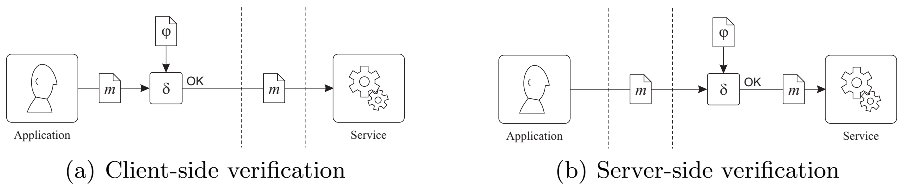
Fig. 4. A sample contract specification. Each constraint is preceded by a caption.

4.2 The BeepBeep Runtime Monitor

Since LTL-FO$^+$ draws heavily on classical LTL, a runtime verification procedure can be obtained from an algorithm presented in [10], which creates the Büchi automaton for a given LTL formula. This algorithm performs on the fly and generates the automaton as the sequence of states unwinds. The LTL-FO$^+$ monitoring procedure, detailed in [15], is an extension of this algorithm, adapted for first-order quantification on message elements.

LTL-FO$^+$ monitoring can then be implemented into a lightweight tool for web applications. It suffices that incoming and outgoing messages be intercepted as "events" and fed to the monitor. The algorithm updates its internal state according to the processed event, and eventually blocks the actual transmission or reception if a violation is discovered.

Since a web application is inherently distributed, the location of this monitor leads to multiple architectural choices, shown in Figure 5. In client-side verification, shown in Figure 5(a), contract compliance is checked in the user's web browser before any message is allowed to be transmitted over the network: an outgoing message m is sent to a function δ monitoring a specification φ. Incoming messages are filtered in the same way before reaching the application's code. Server-side verification 5(b) works on the opposite. A third solution is to use a third-party protocol coordinator (not shown) as suggested by [5]. The coordinator ideally resides neither in the client's browser nor in the web server, and acts as a monitoring proxy for both ends of the communication. To illustrate monitoring on the client side, we developed BeepBeep, a lightweight, Java-based runtime monitor for Ajax web applications.[4]



(a) Client-side verification (b) Server-side verification

Fig. 5. Design choices for runtime verification of web applications

In classical (e.g. Java) programs, intercepting events generally requires instrumenting the code or resorting to mechanisms such as pointcuts [8]. In the present case, network operations converge to a single input-output point, the standard `XMLHttpRequest` object provided by the local browser. It becomes easy to interpose an extra layer of processing over that object, without resorting to any other form of instrumentation.

Including BeepBeep into an existing Ajax application is straightforward. It suffices to host BeepBeep's two files (`beepbeep.jar`, the Java applet, and `beepbeep.js`, an auxiliary JavaScript file) in the same directory as the Ajax application. BeepBeep is bootstrapped by adding a single line in the <`head`> portion of the application's HTML page.

When such a BeepBeep-enabled application is started, the procedure described in Section 2.2 is followed. BeepBeep's additional JavaScript include file dynamically appends the snippet of HTML code instructing the browser to load the Java applet implementing the LTL-FO$^+$ monitoring algorithm, which appears as a small rectangle at the bottom of the application's page. The specification passed to the applet is automatically retrieved from the `contract.txt` file hosted on the server.

The JavaScript code also overloads the methods of the standard `XMLHttp-Request` object. When the original application's JavaScript invokes the `send` method of `XMLHttpRequest`, it actually calls the method implemented by BeepBeep first. This way, incoming and outgoing messages, before being actually sent (or returned), can be deviated to the applet for verification.

4.3 Wrapping Up

We can now return to the Beep Store application and perform runtime monitoring of its interface contract on the client side. Assuming that the store provides

[4] BeepBeep and its source code are available for download under a free software license: `http://beepbeep.sourceforge.net`

a contract file and hosts the two BeepBeep files, we can then modify its HTML code to include the additional JavaScript file, as described above.

The monitor-enabled Beep Store application can be started as usual in a standard browser. As previously, one can open the store's Fault parameters pane, and disable, for example, the internal enforcement of property 3 ("don't login twice"). This time, however, the rectangle at the bottom of the page tells us that BeepBeep successfully fetched a contract and is awaiting for incoming or outgoing XML messages.

The first login attempt can be executed as expected. BeepBeep's display updates, indicating that it indeed witnessed the corresponding messages, but let them through as they did not violate any constraint. After successfully logging in, as expected the faulty client fails to hide the Login link. Clicking on it a second time summons the Login pane, where one can enter the same credentials and press on the Login button. Like before, the client attempts to send a `Login` XML message; however, this time, BeepBeep intercepts the message, correctly discovers that it violates property 3, and skips the piece of code that would normally send it. It also pops a window alerting the user, showing the caption associated with the violated property in the contract file.

This scenario has also been experimented on a real-world web application for the Amazon E-Commerce Service [16]. Our findings indicate that on a low-end computer, monitoring LTL-FO$^+$ contract properties produces an average overhead of around 3% or 10 ms per message in absolute numbers. As a rule, the state of the network accounts for wider variations than the additional processing required by the monitor.

It shall be noted that BeepBeep is independent of any browser-server pair of applications. Its Java applet is self-contained, and the JavaScript auxiliary file can be included into any web page and load it at startup. It can correctly intercept and process any request as long as it is XML-based. Similarly, the contract to be monitored is hosted in a separate text file that is read each time the applet is loaded —hence the contract can be changed without changing the monitor. This way, BeepBeep is a runtime monitoring solution that can be applied to other scenarios than the Beep Store: it suffices to write an appropriate contract for the application under study.

5 Conclusion

This tutorial has highlighted the potential for the application of runtime verification techniques to the field of web services; yet several interesting questions have been left out from this presentation. For example, since events in web applications are sequences of XML messages, it is possible to treat a sequence of such events as one large XML "document" and leverage commercial XML query processors to perform an equivalent validation of message traces [18]. However, the monitoring of quantified formulæ presents a potential for unbounded resource consumption. The *forward-only fragment* of LTL is an ongoing attempt at providing a bounded subset of the logic suitable for limited environments [17].

Finally, if the goal of client-side monitoring is to relieve the server from the burden of dealing with faulty clients, how can one be certain that a client indeed monitors the contract? The concept of *cooperative runtime monitoring* [11] has recently been put forward to resolve such an issue.

Finally, it could be very well possible that application developers refrain from integrating more complex behaviours into their web applications precisely for lack of tools to deal with them in a systematic way. Hence even a modest contribution from runtime verification to the practitioner's toolbox could enhance the quality and ease of development of web applications. In this regard, we hope this tutorial will encourage researchers in the monitoring and validation community to consider web applications as a potential field of application to their work.

References

1. Amazon e-commerce service, `http://solutions.amazonwebservices.com`
2. Paypal web service API documentation, `http://www.paypal.com`
3. Apache Axis (2010), `http://ws.apache.org/axis2`
4. ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24. IEEE Computer Society, Los Alamitos (2010)
5. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services, Concepts, Architectures and Applications, p. 354. Springer, Heidelberg (2004)
6. Barringer, H., Havelund, K., Rydeheard, D.E., Groce, A.: Rule systems for runtime verification: A short tutorial. In: Peled, D. (ed.) RV 2009. LNCS, vol. 5779, pp. 1–24. Springer, Heidelberg (2009)
7. Boutaba, R., Golab, W., Iraqi, Y., Arnaud, B.S.: Lightpaths on demand: A web-services-based management system. IEEE Communications Magazine, 2–9 (July 2004)
8. Chen, F., d'Amorim, M., Roşu, G.: Checking and correcting behaviors of java programs at runtime with Java-MOP. Electr. Notes Theor. Comput. Sci. 144(4), 3–20 (2006)
9. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1, W3C note (2001)
10. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, P., Sredniawa, M. (eds.) PSTV. IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman & Hall, Boca Raton (1995)
11. Hallé, S.: Cooperative runtime monitoring of LTL interface contracts. In: EDOC. IEEE Computer Society, Los Alamitos (to appear, October 2010)
12. Hallé, S., Bultan, T., Hughes, G., Alkhalaf, M., Villemaire, R.: Runtime verification of web service interface contracts. IEEE Computer 43(3), 59–66 (2010)
13. Hallé, S., Ettema, T., Bunch, C., Bultan, T.: Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In: ASE [4] (2010)
14. Hallé, S., Hughes, G., Bultan, T., Alkhalaf, M.: Generating interface grammars from WSDL for automated verification of web services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 516–530. Springer, Heidelberg (2009)
15. Hallé, S., Villemaire, R.: Runtime monitoring of message-based workflows with data. In: EDOC, pp. 63–72. IEEE Computer Society, Los Alamitos (2008)

16. Hallé, S., Villemaire, R.: Browser-based enforcement of interface contracts in web applications with BeepBeep. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 648–653. Springer, Heidelberg (2009)
17. Hallé, S., Villemaire, R.: Runtime monitoring of web service choreographies using streaming XML. In: Shin, S.Y., Ossowski, S. (eds.) SAC, pp. 2118–2125. ACM, New York (2009)
18. Hallé, S., Villemaire, R.: XML query evaluation in validation and monitoring of web service interface contracts. In: dvanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies, pp. 406–424. IGI Global (2010)
19. Hallé, S., Villemaire, R., Cherkaoui, O.: Specifying and validating data-aware temporal web service properties. IEEE Trans. Software Eng. 35(5), 669–683 (2009)
20. Mitra, N., Lafon, Y.: SOAP version 1.2 part 0: Primer, 2nd edn. (2007), `http://www.w3.org/TR/2007/REC-soap12-part0-20070427`