Browser-Based Enforcement of Interface Contracts in Web Applications with BeepBeep

Sylvain Hallé[1] and Roger Villemaire[2]

[1] University of California, Santa Barbara
[2] Université du Québec à Montréal
shalle@acm.org, villemaire.roger@uqam.ca

Abstract. BeepBeep is a lightweight runtime monitor for Ajax web applications. Interface specifications are expressed internally in an extension of LTL with first-order quantification; they can be transparently enforced on the client side 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 reaching the server.

1 Introduction and Motivation

Asynchronous JavaScript and XML (Ajax) refers to a collection of technologies used to develop rich and interactive web applications. A typical Ajax client runs locally in the user's web browser and refreshes its interface using JavaScript according to user input. Popular Ajax applications, such as Google Maps and Facebook, communicate in the background with a remote server; in many cases, the server's functionality is made publicly available as an instance of a *web service*, which can be freely accessed by any third-party Ajax application. These services cannot be invoked arbitrarily: their public documentation specifies constraints on the content of each message and the proper sequence in which they can be exchanged. Yet, nothing prevents an Ajax application from sending messages violating this *interface specification*. In that event, the server can interrupt the communication, reply with an error message, or more insidiously, continue the conversation without warning and eventually send nonsensical or corrupt data. Preventing erroneous or out-of-sequence messages from being sent to the server is desirable, saving bandwidth and allowing client non-conformance to be detected early. To this end, we developed BeepBeep, a lightweight runtime monitor for Ajax applications. BeepBeep's input language is a rich extension of LTL, called LTL-FO$^+$, which includes first-order quantification over message elements and values of a global system clock. By transparently observing the trace of all incoming and outgoing messages inside an Ajax application, BeepBeep can monitor and enforce on-the-fly a wide range of interface contracts, including complex dependencies between message contents, sequences, and time.

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 648–653, 2009.

2 An Example

To illustrate our approach, we consider a library which makes its catalogue available online through a web service interface, allowing users to browse the library catalogue, borrow and return books. Each of these operations can be invoked by sending the proper XML message; for example, the following fragment represents a typical XML message for borrowing a list of books, identified by their book IDs:

```
<message xmlns=''http://example.com/library''>
  <action>borrow</action>
  <books>
    <id>837</id>
    <id>4472</id>
  </books>
</message>
```
(1)

The documentation for the library web service imposes constraints on the messages that can be sent by a client, such as these ones:

1. Every "return" message must precede any "borrow" message
2. Any book can be involved in at most one borrow and/or return operation
3. Any two messages must be at most 60 seconds apart

3 The BeepBeep Runtime Monitor

A standard Ajax application communicates with a web service by sending and receiving messages through the standard `XMLHttpRequest` object provided by the local browser, as shown in the left part of Figure 1. BeepBeep is a lightweight tool that wraps around this object to monitor and enforce interface contracts at runtime (Figure 1, right).[1] The first part of BeepBeep is a small Java applet called the `BeepBeepMonitor`. This applet is responsible for actually keeping track and analyzing the incoming and outgoing messages with respect to an interface contract. The second part is a JavaScript file providing a class called `XMLHttpRequestBB`, which behaves exactly like the standard `XMLHttpRequest`, 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. Any invocations of the original `XMLHttpRequest` object can then be replaced by calls to `XMLHttpRequestBB`. No other changes to the code are required: from this point, BeepBeep intercepts the messages and transparently monitors the conversation.

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

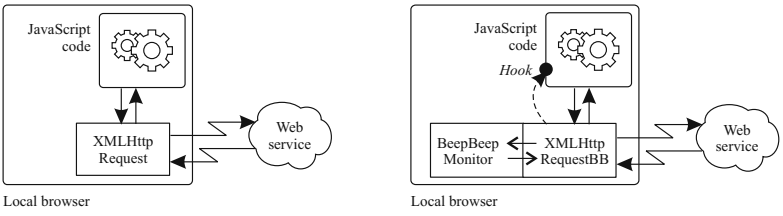

Fig. 1. Standard (left) and BeepBeep-enabled (right) Ajax applications

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.

4 The BeepBeep Contract Specification Language

BeepBeep's language for contract specification is an extension of Linear Temporal Logic called LTL-FO$^+$ [6], whose models are sequences of messages. *Filter expressions* are used to fetch values inside a message; they return a set of values, depending on the *current* message. They are expressed in a subset of the XML Path Language XPath [3]. For example, on the previous XML message (1), the filter `/message/books/id` returns the set $\{837, 4472\}$. *First-order quantifiers* are used to express universal or existential properties over sets returned by filter expressions. Hence the formula $\forall\ x \in$ `/message/books/id` $:\ x \neq$ `123` states that, in the *current* message, no `id` element has value `123`. These quantifiers can then be freely mixed with the traditional *LTL temporal modalities and Boolean connectives* to express complex properties correlating message contents and sequentiality. For example, consider the following formula:

$$\mathbf{G}\,\forall\ a_1 \in \texttt{/message/action}\ :\ (a_1 = \texttt{borrow} \rightarrow \mathbf{X}\ \mathbf{G}\ (\forall\ a_2 \in \texttt{/message/action}\ :\ a_2 \neq \texttt{return}))$$

The formula states that in every message, if the action element has value `borrow`, then from now on, no message can have an action element with value `return`. This is the formal translation of constraint 1 mentioned in Section 2. Quantification can be used to compare values fetched in messages at different points in the trace. For example, the formal translation in LTL-FO$^+$ of constraint 2 stipulates that for any book ID i_1 appearing in a message, then no future message can have some book ID i_2 such that $i_2 = i_1$:

$$\mathbf{G}\,(\forall\ i_1 \in \texttt{/message/books/id}\ :\ \mathbf{X}\,\mathbf{G}\,(\forall\ i_2 \in \texttt{/message/books/id}\ :\ i_1 \neq i_2))$$

Finally, *metric constraints* can be expressed by providing a special filter expression, called "TIME", which always returns a single value: the time of a global

system clock at the moment it is evaluated. Timed properties hence become a special case of quantification, as the translation of constraint 3 shows:

$$\mathbf{G}\,(\forall\, t_1 \in \text{TIME} \;:\; \mathbf{X}\,(\forall\, t_2 \in \text{TIME} \;:\; t_2 - t_1 < 60))$$

5 Tool Highlights and Related Work

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, which is generic. This is in contrast with [7, 9, 11], 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 monitoring plan is generated automatically from any LTL-FO$^+$ formula passed to the monitor.

This dynamicity is possible thanks to BeepBeep's monitoring algorithm. It is based on an algorithm presented in [5], 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. Although LTL monitoring requires exponential space [12], in practice the on-the-fly algorithm generates a subset of the automaton with negligible space. BeepBeep's monitoring procedure, detailed in [6], is an extension of this algorithm, adapted for LTL-FO$^+$'s first-order quantification on message elements. It includes a number of optimizations, such as the simplification of unsatisfiable subformulæ and the use of three-valued logic [2] to allow for "inconclusive" trace prefixes.

BeepBeep distinguishes itself from related work in a number of aspects:

- Message monitoring. 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.
- Rich input language. BeepBeep's LTL-FO$^+$ allows first-order quantification over XPath expressions to fetch values inside messages, store them and compare them at a later time. Contrarily to similar message-based logics, such as LTL-FO [4], there is no restriction on the use of temporal operators inside quantifiers, and vice versa. BeepBeep can handle arbitrary nested structures; no upper bound on the arity of the messages needs to be fixed in advance.
- Client-side monitoring. Erroneous messages are trapped at the source, saving bandwidth and CPU time on the server. This contrasts with [10, 1] where monitoring of the conversation is done on the server side.
- Non-invasive. Runtime monitoring of arbitrary interface contracts can be enforced transparently with minor changes to the code, apart from including BeepBeep. Other approaches, such as [9], require heavier code instrumentation in order to correctly intercept non-compliant behaviour.

– 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.
– Universal. BeepBeep works off-the-shelf on any browser supporting Java applets, including proprietary software with closed source code. It does not require a modified version of the browser (hooks) to work, as is the case for Browser-enforced Embedded Policies (BEEP) described in [8].

6 Experimental Evaluation

BeepBeep has been tested on Ajax applications in various scenarios.[2] 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 and plotted in Figure 2. 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 (explaining the negative values). Our findings indicate that on low-end computer (Asus EeePC with a 600 MHz processor), monitoring LTL-FO$^+$ contract properties produces an average overhead of around 3%, a negligible 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.

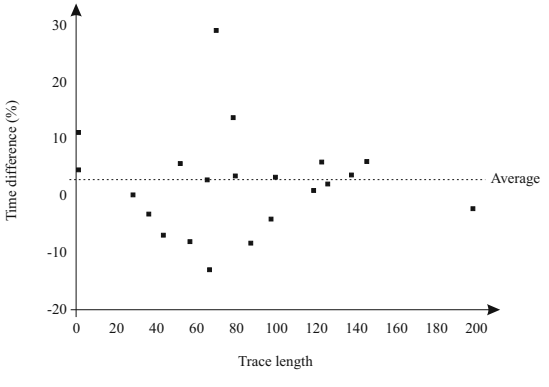

Fig. 2. Overhead for a BeepBeep-enabled Ajax client

Therefore, we conclude that BeepBeep can efficiently monitor LTL-FO$^+$ runtime properties at a very low cost. By providing a transparent and very simple

[2] See http://beepbeep.sourceforge.net/examples for more details.

way of enforcing rich interface contracts into virtually any existing Ajax application, BeepBeep contributes to increase the reach of logic and formal verification approaches in the development of everyday web applications.

References

1. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-time monitoring of instances and classes of web service compositions. In: ICWS, pp. 63–71. IEEE Computer Society, Los Alamitos (2006)
2. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 260–272. Springer, Heidelberg (2006)
3. Clark, J., DeRose, S.: XML path language (XPath) version 1.0, W3C recommendation (1999)
4. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven web services. In: Deutsch, A. (ed.) PODS, pp. 71–82. ACM, New York (2004)
5. 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)
6. Hallé, S., Villemaire, R.: Runtime monitoring of message-based workflows with data. In: EDOC, pp. 63–72. IEEE Computer Society Press, Los Alamitos (2008)
7. Hughes, G., Bultan, T., Alkhalaf, M.: Client and server verification for web services using interface grammars. In: Bultan, T., Xie, T. (eds.) TAV-WEB, pp. 40–46. ACM, New York (2008)
8. Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: Williamson, C.L., Zurko, M.E., Patel-Schneider, P.F., Shenoy, P.J. (eds.) WWW, pp. 601–610. ACM, New York (2007)
9. Krüger, I.H., Meisinger, M., Menarini, M.: Runtime verification of interactions: From MSCs to aspects. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 63–74. Springer, Heidelberg (2007)
10. Mahbub, K., Spanoudakis, G.: Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In: ICWS, pp. 257–265. IEEE Computer Society, Los Alamitos (2005)
11. Rosu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: This time with calls and returns. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 51–68. Springer, Heidelberg (2008)
12. Rosu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. 12(2), 151–197 (2005)