Runtime Verification of Web Service Interface Contracts

Sylvain Hallé, Tevfik Bultan, Graham Hughes, Muath Alkhalaf,
University of California, Santa Barbara
Roger Villemaire, Université du Québec à Montréal

Web applications are required to follow an interface contract that specifies their expected behaviour when they communicate with a web service. Using the Amazon E-Commerce Service as an example, we show how we can automatically test an implementation for conformance as well as monitor at runtime that each partner fulfils its part of the contract.

The term *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 on-the-fly in response to user input. Popular Ajax applications, such as Google Maps and Facebook, communicate in the background with a server: entering information in the Facebook portal sends it to its remote database; dragging Google's map triggers the retrieval of new portions of the image from their server.

In many cases, the server's functionality is made publicly available as an instance of a *web service* and can be freely accessed by any third-party Ajax application. However, this appealing modularity is also 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? Whether for specifying interoperability constraints, business policies or legal guidelines, a good web service has to have a well defined and enforceable *interface contract* [1].

THE AMAZON E-COMMERCE SERVICE

Take the example of a popular suite of web services, the Amazon Web Services (AWS). AWS provides a diverse roster of services, among them the Amazon Associates Web Service (also known as the Amazon E-Commerce Service or "ECS"), a free service that exposes Amazon's product data with the goal of driving traffic back to Amazon's web site. This access is provided through a SOAP interface specified with WSDL (sidebar). It lists 40 operations, almost all of which are differing ways of searching Amazon's product database.

We built an Ajax application using the AWS-ECS API, and in particular a core of six operations.

- `ItemSearch` searches Amazon's database for items matching some keywords. It returns a list of products that match the search criteria.
- `CartCreate` takes an ASIN —a unique identifier for that item in Amazon's database— and a positive integer *n* and creates a new shopping cart that represents a request for *n* copies of that item. The cart's contents is stored and managed by Amazon, the SOAP operations refer to the cart by passing its unique ID.
- `CartGet` returns the content of a given cart.
- `CartAdd` adds a new row to a given cart requesting *n* copies of some ASIN.
- `CartModify` is used to change the quantity of some item in a given cart (0 deletes it).
- `CartClear` empties the cart with given ID.

Sidebar: Defining a Web Service

Web services interact with each other by exchanging messages encoded using the eXtensible Markup Language (XML). In its basic form, the web service architecture consists of a simple RPC model where a client invokes operations exported by a service provider using the Simple Object Access Protocol (SOAP), a standard communication protocol for transmitting XML messages. Each web service has to publish its invocation interface, e.g., network address, ports, operations provided, and the expected XML message formats to invoke the service, using the Web Services Description Language (WSDL). The WSDL specification serves as a contract between the client and the server that (in part) defines the valid interactions.

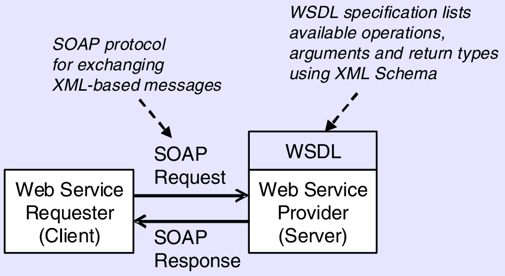


Figure 1 (left) shows our own Ajax client, which serves as a thinly veiled interface to the AWS-ECS methods, using XML messages sent through SOAP to invoke them. A possible sequence of operations is shown on the right of Figure 1.

THE AWS-ECS INTERFACE CONTRACT

Like any other web service, the AWS-ECS cannot be invoked arbitrarily. When developing our client, we had to peruse its WSDL specification, where the structure of each XML message sent and received is defined. We had to make sure that our client produced messages following these conventions.

This, however, is not sufficient. In addition, we found that the online public documentation gives other requirements, expressed in plain English, that WSDL is not designed to cover.[1]

Data Constraints

We first found data constraints, i.e. limitations on the structure and content of the messages sent to the service. For example:

1. In the `ItemLookup` message, the element `MerchantID` is optional, unless the element `IdType` has value `SKU`, in which case it becomes mandatory.

2. In the `ItemSearch` message, the element `ItemPage` only accepts integers between 1 and 400. The WSDL only specifies it as a positive integer without giving any bound.

Control-Flow Constraints

Although web services were originally intended to be stateless, the AWS-ECS allows long-running transactions whose state is carried through manipulations of the shopping cart. Therefore, we found several restrictions on the *sequence* in which operations are invoked:

3. Except for `ItemSearch` and `CartCreate`, every operation requires that a cart already exists, e.g. that they follow a `CartCreate` operation.

4. `CartModify` can only be called after an item identifier has been retrieved from `CartCreate` or `CartAdd`.

Data-aware Control Flow Constraints

Worse yet, the AWS-ECS imposes control flow constraints that are also *data-aware*, i.e. the authorized sequence of messages depends on relationships between values of multiple messages. For example:

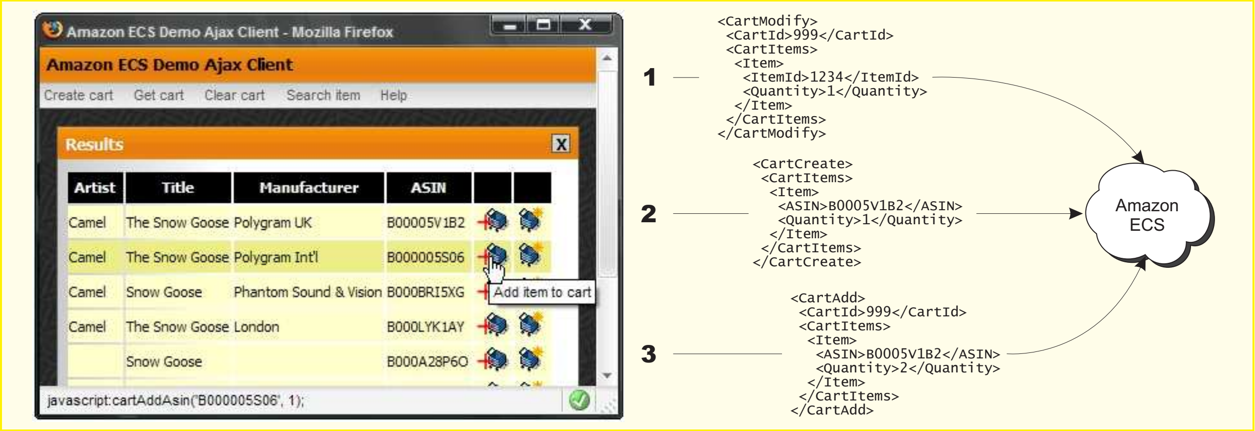
5. One `CartModify` after another on the same item will fail if the first one deleted the item.

6. Even though there can only be one row for each ASIN, CartModify only takes the item identifier returned by CartAdd or CartCreate.

CONTRACT VIOLATIONS

Alas, our Ajax client, typical of many others, does not take care of most of these constraints. It is purposefully “faulty” by allowing any ECS operation to be invoked in any order, and fails to automatically refresh the contents of a cart when modifications are made to it. Therefore, nothing prevents a user from using the interface to send messages violating these *interface specifications*. For example, one can see that the sequence of messages in Figure 1 is illegal: first, a `CartModify` message is sent before a `CartCreate`, and second, the `CartAdd` operation attempts to add an item already present in the cart. Note that each individual message fulfils the description given in the WSDL of the AWS-ECS, as we had made sure of. The violation rather comes from the relationship between these messages: their respective values, and their position in the sequence.

[1] All constraints are extracted from the online documentation for the AWS-ECS, http://docs.amazonwebservices.com/-AWSEcommerceService/2007-01-17/



Artist	Title	Manufacturer	ASIN		
Camel	The Snow Goose	Polygram UK	B00005V1B2		
Camel	The Snow Goose	Polygram Int'l	B000005S06		
Camel	Snow Goose	Phantom Sound & Vision	B000BRI5XG		
Camel	The Snow Goose	London	B000LYK1AY		
	Snow Goose		B000A28P6O		

```
javascript:cartAddAsin('B000005S06', 1);
```

```
<CartModify>
 <CartId>999</CartId>
 <CartItems>
  <Item>
   <ItemId>1234</ItemId>
   <Quantity>1</Quantity>
  </Item>
 </CartItems>
</CartModify>
```

```
<CartCreate>
 <CartItems>
  <Item>
   <ASIN>B0005V1B2</ASIN>
   <Quantity>1</Quantity>
  </Item>
 </CartItems>
</CartCreate>
```

```
<CartAdd>
 <CartId>999</CartId>
 <CartItems>
  <Item>
   <ASIN>B0005V1B2</ASIN>
   <Quantity>2</Quantity>
  </Item>
 </CartItems>
</CartAdd>
```

Figure 1: An Ajax client and a sequence of three XML messages it can send to the AWS-ECS. Some message elements and the responses from the AWS-ECS have been omitted.

If a message, or sequence of messages violates the interface contract with a web service, several things can happen. 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. Moreover, there exists the possibility that the server's implementation does not match its documentation.

Currently, the widely used remedy is for the application developer to read through the documentation, and incorporate appropriate constraints into the control flow of the application, so that erroneous messages or message sequences are intercepted or simply cannot be produced — this should indeed be the case for any robust application. Conformance to the contract cannot be guaranteed, but can be observed through testing, using for example Yahoo's YUI Test Utility.[2] In any case, deciding what and how to test still has to be done manually, and no insurance can be given on the coverage of the tests with respect to the specification.

We chose to follow an alternate, or rather complementary approach, consisting of building a well-defined description of the interaction between the Ajax application and the web service, expressed in some formal language. This *model* of the expected behaviour then lends itself to several analysis techniques; for example, when both the client and the server belong to the same organization, a first possibility is to perform static analysis. The client-server pair can be studied as a single system whose possible execution traces are systematically searched for violations of the interface contract —this is called *model checking* [2]. Since the model is formal, the analysis is often automated and does not require user intervention.

Yet, in many cases (as in ours), the application developer is not part of the organization providing the web service to be accessed. Therefore, the service appears as a "black box", a component opaque to model checking. Formal analysis is still possible, but at runtime, using the real implementation of the web service. This is achieved with the help of a *protocol controller*, which possesses a formal model of the interface contract and updates its model according to the runtime behaviour of the client and the service (Figure 2(a)), as their conversation unfolds.

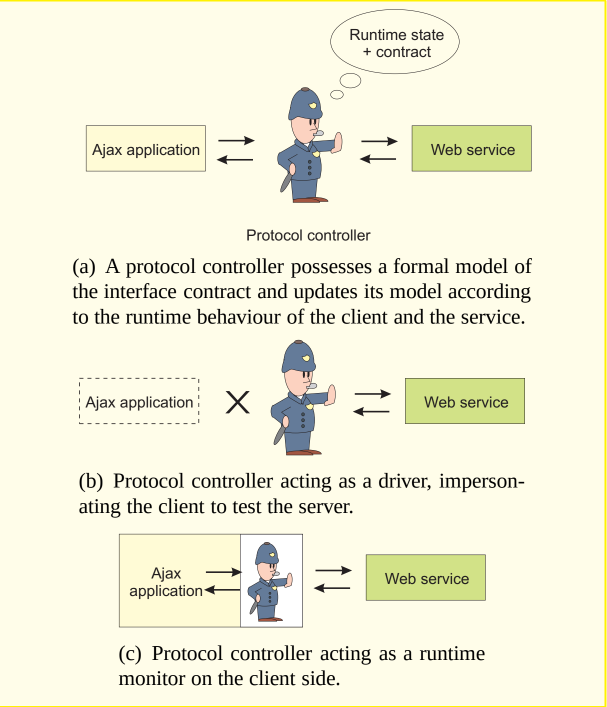
RUNTIME SERVER TESTING

To discover whether we correctly understand the service implementation, and also to check that the web service actually fulfils the constraints mentioned in its own documentation, a first solution is to perform *runtime testing*. In this situation, a protocol controller uses the interface specification to impersonate one of the peers and communicate with the other. When the controller stands for the client side of the communication, it is called a *driver* (Figure 2(b)). A driver generates a sequence of invocations that a possible contract-compliant client could send to the service. This sequence is fed to the actual service instance, and the response from the service is then checked for compliance with the contract. If the driver does not receive what its contract expects, it signals a mismatch between this formal contract (which represents what we believe is the appropriate behaviour) and the actual server implementation.

A Runtime Model of the AWS-ECS

We implemented this idea by automatically generating a web service driver, using interface grammars as the

[2] http://developer.yahoo.com/yui/yuitest



(a) A protocol controller possesses a formal model of the interface contract and updates its model according to the runtime behaviour of the client and the service.

(b) Protocol controller acting as a driver, impersonating the client to test the server.

(c) Protocol controller acting as a runtime monitor on the client side.

Figure 2: Various uses of a protocol controller at runtime.

Sidebar: Interface Grammars

An interface grammar is expressed as a series of productions of the form $a(v_1,\ldots,v_n) \rightarrow A$. Here $v_1,\ldots,v_n$ correspond to the parameters of the non-terminal a and A is the right hand side of the production which may contain the following:

- nonterminals, which we express as $nt(v_1,\ldots,v_n)$;
- semantic predicates that must evaluate to true for the production to be available, which we express as p;
- semantic actions that are executed during the parse, which we express as $\langle\langle a\rangle\rangle$;
- outgoing method calls, which we express as $!m(v_1,\ldots,v_n)$;
- returns from outgoing method calls, which we express as ¡$m(v_1,\ldots,v_n)$.

As with any grammar, the successive application of derivation rules to non-terminal symbols can be used to generate sentences or to parse an existing sentence for correctness. In the present case, these sentences represent sequences of web service operations.

specification language (sidebar). Compared to many existing specification languages for web services, the novelty of these grammars is their ability to model complex behaviours mixing control flow *and* data. For the needs of the presentation, the interface grammar shown in Figure 3 represents the client interface for a simplified version of the AWS-ECS service (our experiments were performed on a more detailed specification).

A compiler takes this interface grammar and automatically generates Java code for a driver —that is, a language generator that produces SOAP request sequences based on the grammar [3]. The sentence generating algorithm is basically the same, whatever the grammar: it begins with the start symbol and generates a derivation by applying a randomly chosen production rule to some non-terminal symbol, until no such symbol remains. At the same time as the symbols are produced, the corresponding XML messages are sent to the web service.

Results

To test the AWS-ECS, we ran the generator until it produced 100 sentences, which were executed on the actual service implementation as they were built. Testing each sentence took an average of 3 seconds. All the non-terminal symbols in the grammar were covered after generating an average of 41 sentences, or roughly 2 minutes of testing. The average number of steps in derivations generated by the random generator was 17.5, and the average number of SOAP requests that were generated per derivation was 3.2.

Performing these server verification experiments were useful, as they helped us discover two errors; the first is a mismatch between the server implementation and its documentation, while the second results from our misinterpretation of the documentation.

- Error 1: We found that it is illegal to send a `CartAdd` for an item's ASIN which is already in the cart. Instead of creating two lines with distinct item IDs, or updating the existing line by adding its quantity to the current request, the AWS-ECS replies with an error message. This restriction is not explicitly stated in the AWS-ECS API specification.

- Error 2: Our driver checks that the contents of the cart returned by Amazon are precisely those we expect to see. We thought that an empty cart would have a non null `CartItems` element that contains an array of zero length. Instead, the returned cart has no `CartItems` element at all. This issue is not explicitly stated in the API documentation either, although it is present in the WSDL specification.

$$
\begin{array}{rcl}
\mathit{start} & \rightarrow & \mathit{search}(\text{asin}); \mathit{cart}(\text{asin}) \\
 & | & \varepsilon \\
\mathit{search}(\text{asin}) & \rightarrow & !\textsc{ItemSearch}(); \text{¡}\textsc{ItemSearch}(\text{asin}); \mathit{search}'(\text{asin}) \\
\mathit{search}'(\text{asin}) & \rightarrow & !\textsc{ItemSearch}(); \text{¡}\textsc{ItemSearch}(\text{asin}); \mathit{search}'(\text{asin}) \\
 & | & \varepsilon \\
\mathit{cart}(\text{asin}) & \rightarrow & !\textsc{CartCreate}(\text{asin}); \text{¡}\textsc{CartCreate}(\text{cart}, \text{item}); \mathit{permute}(\text{cart}, \text{item}); \mathit{clear}(\text{cart}) \\
 & | & \varepsilon \\
\mathit{permute}(\text{cart}, \text{item}) & \rightarrow & !\textsc{CartGet}(\text{cart}); \text{¡}\textsc{CartGet}(\text{cart}); \mathit{permute}(\text{cart}, \text{item}) \\
 & | & \langle\langle \textsc{choose}\, n > 0 \rangle\rangle; !\textsc{CartModify}(\text{cart}, \text{item}, n); \text{¡}\textsc{CartModify}(\text{cart}); \mathit{permute}(\text{cart}, \text{item}) \\
 & | & !\textsc{CartModify}(\text{cart}, \text{item}, 0); \text{¡}\textsc{CartModify}(\text{cart}) \\
 & | & \mathit{search}(\text{asin}); \mathit{permute}'(\text{cart}, \text{asin}); \mathit{permute}(\text{cart}, \text{item}) \\
 & | & \varepsilon \\
\mathit{permute}'(\text{cart}, \text{asin}) & \rightarrow & \llbracket \text{asin} \notin \text{ran}(\text{cart}) \rrbracket; !\textsc{CartAdd}(\text{cart}, \text{asin}); \text{¡}\textsc{CartAdd}(\text{cart}, \text{item}); \mathit{permute}(\text{cart}, \text{item}) \\
 & | & \varepsilon \\
\mathit{clear}(\text{cart}) & \rightarrow & !\textsc{CartClear}(\text{cart}); \text{¡}\textsc{CartClear}(\text{cart}); \mathit{clear}(\text{cart}) \\
 & | & !\textsc{CartGet}(\text{cart}); \text{¡}\textsc{CartGet}(\text{cart}); \mathit{clear}(\text{cart}) \\
 & | & \mathit{search}(\text{asin}); \mathit{clear}'(\text{cart}, \text{asin}) \\
 & | & \varepsilon \\
\mathit{clear}'(\text{cart}, \text{asin}) & \rightarrow & \llbracket \text{asin} \notin \text{ran}(\text{cart}) \rrbracket; !\textsc{CartAdd}(\text{cart}, \text{asin}); \text{¡}\textsc{CartAdd}(\text{cart}, \text{item}); \\
 & & \mathit{permute}(\text{cart}, \text{item}); \mathit{clear}(\text{cart}) \\
 & | & \varepsilon
\end{array}
$$

Figure 3: Interface grammar for an AWS-ECS client

RUNTIME MONITORING OF AJAX APPLICATIONS

Our interface driver is able to detect contract non conformance of the service through the generation of runtime test sequences, when the target service is known in advance. However, in service-oriented architectures, partners can be discovered dynamically: in such a situation, neither static analysis nor runtime testing can be done in advance. In any case, it might be desirable that some form of "safety net" be still present during the normal execution of the client, for various reasons. For example, 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 [4]. As a second step to our work, we wanted to dynamically detect any contract violations from either side as they happen.

In this situation, the actual Ajax application communicates with the actual service. The protocol controller acts as a *runtime monitor* and silently eavesdrops the sequence of exchanged messages between the application and the service. It acts only when a contract violation is detected, by raising an error and/or blocking the communication.

The BeepBeep Runtime Monitor

The monitor can be hosted inside the client (Figure 2(c)), inside the server, or in a module independent of each. Contrarily to existing approaches [5, 6], we chose to implement a monitor on the client side, believing that erroneous behaviour is more likely to come from untested clients than from a production server. This way, erroneous messages are trapped at the source, saving bandwidth and CPU time on the server. To this end, we developed BeepBeep, a lightweight runtime monitor for Ajax web applications.[3]

BeepBeep's architecture has been designed to be minimally invasive. A standard Ajax application communicates with a web service by sending and receiving messages through the `XMLHttpRequest` object provided by the local browser. BeepBeep wraps around this object through a JavaScript file providing a class called `XMLHttpRequestBB`. It behaves exactly like the original, with the exception that incoming and outgoing messages, before being actually sent (or returned), are analyzed and possibly blocked if violations are found. This follows a

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

principle already used to prevent script injection attacks [7], although here no modification to the browser is required. Rather, a small and invisible Java applet called the `BeepBeepMonitor` is responsible for actually analyzing the incoming and outgoing messages with respect to the interface contract, and signalling eventual errors back to the JavaScript code on-the-fly, following an algorithm we devised [8].

To add BeepBeep to our (or any existing) Ajax application, the JavaScript file and Java applet (less than 50 kb in total) are copied to an arbitrary location on the application's host server. The only code instrumentation required is to include BeepBeep by adding a single line of code in the <`head`> portion of our application's HTML page. From now on, any call to `XMLHttpRequest` need simply to be replaced by a call to `XMLHttpRequestBB` in order for BeepBeep to intercept and monitor the conversation.

While an interface grammar was well suited for our sentence generating driver, its monolithic nature makes it hard to identify what element of the contract becomes violated when an error occurs. For the needs of runtime monitoring, BeepBeep is fed with a list of independent *constraints* which, taken together, form the contract specification to enforce. These constraints are expressed in a language called LTL-FO$^+$, an extension of the well-known Linear Temporal Logic (LTL) to accommodate first-order quantification over message elements and values of a global system clock (sidebar). Each constraint can be violated (or fulfilled) independently of the others, making it easier to display informative error messages to the user. Figure 4 shows a contract file for BeepBeep containing three of AWS-ECS's interface constraints given as LTL-FO$^+$ expressions. This contract is located on the server side in a file separate from the monitor itself. Changing the contract can be done dynamically without making changes to the clients or the monitor itself. This is in contrast with [9, 10], 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 runtime web service interactions.

Results

We tested BeepBeep on our Ajax client for the AWS-ECS. As we now know, the client enables a user to perform a wide range of contract violations: get a cart before creating a cart, add or remove the same item twice; in total we provided BeepBeep with 11 such possible violations to watch.[4] By using the client and producing non-compliant sequences of operations for all 11 prop-

Sidebar: LTL-FO$^+$

LTL-FO$^+$ is a logic on sequences of XML messages. It provides all the familiar Boolean connectives: & ("and"), $\|$ ("or"), ! ("not"), $\rightarrow$ ("if then"); they can be used to combine expressions as usual. Data inside a message can be accessed using first-order quantifiers, and sequences of messages are constrained using the following operators:

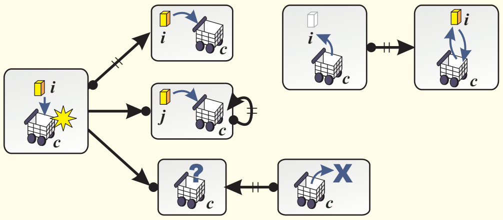
- $[x \in \pi]$ means "for every x in π". Here, x is a variable, and π is a path expression used to fetch possible values for x. For example, if φ is a formula where x appears, then $[x \in /tag1/tag2] : \varphi$ means: every value at the end of /tag1/tag2 satisfies φ. Similarly, $\langle x \in \pi \rangle$ means "some x in π".
- $\mathbf{X}$ means "in the next message". For example, if φ is a formula, $\mathbf{X}\,\varphi$ says that φ will be true in the next message. Similarly, $\mathbf{G}$ means "globally" (for all messages) and $\mathbf{F}$ means "eventually" (for at least one message in the future).
- $\mathbf{U}$ means "until". If φ and ψ are formulæ, writing $\varphi\,\mathbf{U}\,\psi$ says that ψ will be true eventually, and in the meantime, φ is true for every message until ψ becomes true.

```
% All ItemLookup messages must have a MerchantID
  element if their IdType is "SKU"
; G (([i /ItemLookup/IdType] ((i) = (SKU))) ->
    (<m /ItemLookup/MerchantId> (TRUE)))

% No CartModify can appear once a CartClear has
  been issued
; G ((<i /CartClear> (TRUE)) ->
    (G (!(<j /CartModify> (TRUE)))))

% It is illegal to add an ASIN to a cart that
  already has a row for that ASIN
; G ([i /CartAdd/List/ASIN] (X (G
    ([j /CartAdd/List/ASIN] ((i) = (j))))))
```

(a)



(b)

Figure 4: (a) A sample BeepBeep contract specification. Each constraint, expressed in LTL-FO$^+$, is preceded by a caption that is displayed by the application when it becomes violated. (b) A similar set of constraints, expressed graphically in a notation called DecSerFlow.

[4] The demo is available on BeepBeep's website: http://beepbeep.sourceforge.net/examples/amazon-ecs.

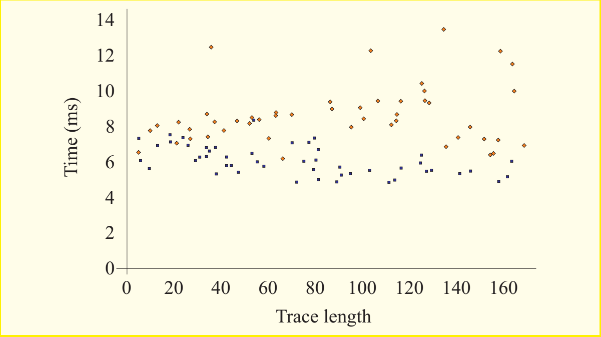


Figure 5: Monitoring time per message for various trace lengths, with data domains of 25 (blue) and 200 (orange) elements.

erties, we observed that BeepBeep caught all of these violations instantly, as they were produced, by popping up an alert message and preventing the faulty messages from reaching Amazon. The monitor works in a bidirectional fashion and also looks for specification mismatches from the server side. Hence, it correctly diagnosed Error 2, found earlier by our runtime tester, by warning the user that empty carts did not contain a `CartItem` container.

To measure the performance overhead induced by the addition of a runtime monitor, we programmed the client to generate 100 random message traces of length ranging from 5 to 200 messages. For half of these traces, the requests were created from a pool of 25 randomly selected items; for the other half, the requests were created from a pool of 200 items. Each of these traces was then "played back" to the AWS-ECS, as if a real user were using the interface. For each of these traces, we measured the total CPU time spent for the execution of the BeepBeepMonitor Java applet, and plotted the results in Figure 5.

The time required to process a message to (or from) the AWS-ECS remains well under 20 milliseconds, taking in average 8 ms, a figure that was not significantly modified by increasing the pool of possible items in the requests by a factor 8. The length of the transaction (i.e. the total number of messages) has little effect on the monitor's performance. This added roughly 3% to the total round-trip time of a request-response to Amazon. Moreover, these times were computed on a low-end EeePC netbook running at 600 MHz.

WHO SHOULD WRITE THE CONTRACTS?

While our experiments at testing and monitoring interface contracts were successful, to this day the *extraction* of such contracts remains mostly a manual task. Moreover, the question remains of *who* should provide these contracts and *where* they should be obtained from. The current state of our research does not provide definitive solutions to these issues. However, we can mention ongoing work on these topics pointing to promising solutions.

Alternative Specification Languages

Interface grammars and temporal logic are appropriate representations for the automated processing of interface contracts. However, finite state machines, regular expressions and UML activity diagrams are all appropriate (and, to some extent, equivalent) representations of interface contracts. Most of them can be easily translated into equivalent grammars or temporal logic expressions.

Moreover, in most situations, interface contracts can be expressed by combining a small set of predefined templates or *patterns*. An example is the DecSerFlow language [11], which uses a graphical notation to express simple sequential relationships. Figure 4(b) shows a screenshot of a prototype tool we are currently developing, where the AWS-ECS contract, input in DecSerFlow notation, is internally converted into LTL-FO$^+$ and displayed graphically to the user. Each box depicts a specific message. An arrow A→B indicates that A must occur before B; a barred arrow rather indicates that once A occurs, B cannot occur. The top-left arrow therefore expresses that once a cart c is created with an item i, one can no longer add that same item i to that same cart c.

Other solutions focus on extracting these constraints automatically, either by analyzing the service's source code [12], or by using machine learning techniques on sample execution traces [13].

Providing the Contract

Developers are already accustomed to the practice of defining and providing an interface contract for the web services they create: after all, the WSDL document that accompanies a service *is* an interface contract. However, as we pointed out earlier, WSDL covers only part of the requirements for a successful interaction with a service; in particular, it lacks a formal specification of allowed sequences of messages and related constraints on data parameters.

Eventually, service developers could cultivate the habit of documenting such constraints in the same way WSDL message definitions currently are, and could warn potential users of their presence by storing them in a machine-readable form at a similar URI. Message Exchange Patterns, introduced in WSDL 2.0, represent a step in that direction, although these patterns can include at most two messages and do not seem sufficient for long-running transactions such as those we studied in the AWS-ECS.

In retrospect, our experiments with the Amazon E-Commerce Service (AWS-ECS) showed us the advantages of using a model-based approach for the runtime testing and monitoring of web applications. Once a formal model of the runtime behaviour of a web service is created, we can perform a variety of analysis tasks without user intervention.

In a first step, we can verify our understanding of the documentation by automatically producing test sequences that are run on the actual implementation of the web service. Then, our runtime tool BeepBeep is able to seamlessly and easily enforce interface contracts on the client side, warning the user of violations and preventing erroneous messages from reaching their destination.

Doing so with the AWS-ECS enabled us to: 1) automatically generate test sequences and detect two deviations of their service implementation with respect to the online documentation provided, in less than three minutes of testing; 2) provide a framework that allows the runtime monitoring of both client and server contract constraints with minimal modification to an existing Ajax application's code, with an associated overhead of roughly 10 milliseconds for each incoming or outgoing message. The tools we developed are generic and our approach can be used on any other client or service.

References

[1] Greg Meredith and Steve Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.

[2] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000. ISBN 0-262-03270-8. 330 pp.

[3] Graham Hughes, Tevfik Bultan, and Muath Alkhalaf. Client and server verification for web services using interface grammars. In Tevfik Bultan and Tao Xie, editors, *TAV-WEB*, pages 40–46. ACM, 2008. ISBN 978-1-60558-053-1.

[4] William N. Robinson. Monitoring web service requirements. In *RE*, pages 65–74. IEEE Computer Society, 2003. ISBN 0-7695-1980-6.

[5] Khaled Mahbub and George Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS*, pages 257–265. IEEE Computer Society, 2005. ISBN 0-7695-2409-5.

[6] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS*, pages 63–71. IEEE Computer Society, 2006. ISBN 0-7695-2669-1.

[7] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *WWW*, pages 601–610. ACM, 2007. ISBN 978-1-59593-654-7.

[8] Sylvain Hallé and Roger Villemaire. Runtime monitoring of message-based workflows with data. In *EDOC*, pages 63–72. IEEE Computer Society, 2008.

[9] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. Runtime verification of interactions: From MSCs to aspects. In Oleg Sokolsky and Serdar Tasiran, editors, *RV*, volume 4839 of *Lecture Notes in Computer Science*, pages 63–74. Springer, 2007. ISBN 978-3-540-77394-8.

[10] Grigore Rosu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties: This time with calls and returns. In Martin Leucker, editor, *RV*, volume 5289 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2008. ISBN 978-3-540-89246-5.

[11] Wil M.P. van der Aalst and Maja Pesic. DecSerFlow: Towards a truly declarative service flow language. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2006. ISBN 3-540-38862-1.

[12] William G.J. Halfond and Alessandro Orso. Improving test case generation for web applications using automated interface discovery. In Ivica Crnkovic and Antonia Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 145–154. ACM, 2007. ISBN 978-1-59593-811-4.

[13] Alex Groce, Klaus Havelund, and Margaret Smith. Let's look at the logs: Low-impact runtime verification. In *COMPASS*, 2009.

Sylvain Hallé *is a postdoctoral researcher at University of California, Santa Barbara. He received a PhD in computer science from Université du Québec à Montréal. Contact him at shalle@acm.org.*

Tevfik Bultan *is a professor of computer science at University of California, Santa Barbara. He received a PhD in computer science from the University of Maryland, College Park. Contact him at bultan@cs.ucsb.edu.*

Muath Alkhalaf *is a PhD student in computer science at University of California, Santa Barbara. Contact him at muath@cs.ucsb.edu.*

Graham Hughes *received a PhD in computer science from the University of California, Santa Barbara. Contact him at graham@cs.ucsb.edu.*

Roger Villemaire *is a professor of computer science at Université du Québec à Montréal. He received a PhD in mathematics from University of Tübingen. Contact him at villemaire.roger@uqam.ca.*