Generating Interface Grammars from WSDL for Automated Verification of Web Services *

Sylvain Hallé, Graham Hughes, Tevfik Bultan, and Muath Alkhalaf

University of California
Santa Barbara, CA 93106-5110 USA
shalle@acm.org, {graham,bultan,muath}@cs.ucsb.edu

Abstract. Interface grammars are a formalism for expressing constraints on sequences of messages exchanged between two components. In this paper, we extend interface grammars with an automated translation of XML Schema definitions present in WSDL documents into interface grammar rules. Given an interface grammar, we can then automatically generate either 1) a parser, to check that a sequence of messages generated by a web service client is correct with respect to the interface specification, or 2) a sentence generator producing compliant message sequences, to check that the web service responds to them according to the interface specification. By doing so, we can validate and generate both messages and sequences of messages in a uniform manner; moreover, we can express constraints where message structure and control flow cannot be handled separately.

1 Introduction

Service-oriented architecture (SOA) has become an important concept in software development with the advent of web services. Because of their flexible nature, web services can be dynamically discovered and orchestrated to form value-added e-Business applications. However, this appealing modularity is the source of one major issue: while dynamically combining cross-business services, how can one ensure the interaction between each of them proceeds as was intended by their respective providers? Achieving modularity and interoperability requires that the web services have well defined and enforceable interface contracts [20].

Part of this contract is summarized in the service's WSDL document, which specifies its acceptable message structures and request-response patterns. This document acts as a specification that can be used both to *validate* and to *generate* messages sent by the client or the service. This double nature of WSDL makes it possible to automatically produce test requests validating the functionality of a service, or to test a client by communicating with a local web service stub that generates WSDL-compliant stock responses.

As it is now well known, many web services, and in particular e-commerce APIs such as the Amazon E-Commerce Service, Google Shopping or PayPal,

* This work is supported by NSF grants CCF-0614002 and CCF-0716095.

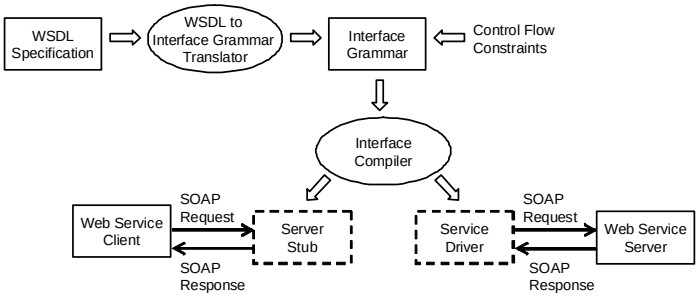

Fig. 1: Our web service verification framework

introduce the notion of *sessions* and constrain communications over several request-response blocks. The previous approach does not generalize to such scenarios. Apart from attempts at validation of service interactions through runtime monitoring of message sequences [5,6,13,14,18], for the most part the question of *generating* a control-flow compliant sequence of messages, for simulation, testing or verification purposes, remains open.

Interface grammars are a specification formalism that has been proposed to enable modular verification [16]. An interface grammar specifies the allowable interactions between two components by identifying the acceptable call/return sequences between them. In earlier work, we proposed their use for expressing the control-flow constraints on a client interacting with a web service [17]. However, message elements in these grammars were regarded as terminal symbols; to actually generate or validate a given message, hand-coded Java functions had to be written and hooked to their respective grammar counterpart. In this paper, we bridge the gap between control flow and message specifications by developing an automated translation of WSDL documents into interface grammar rules.

In Section 2, we present a real-world web service, the PayPal Express Checkout API. We exhibit constraints where the sequence of allowed operations and their data content are correlated, and express them with the use of interface grammars.

Our web service verification framework (Figure 1) consists of two tools: 1) a WSDL-to-interface grammar translator and 2) an interface compiler. First, the WSDL to interface grammar translator takes a WSDL specification as input and converts it into an interface grammar; this translation is described in Section 3. Constraints that are not expressed in WSDL (such as control-flow constraints) can then be added to this automatically generated interface grammar.

In Section 4, we use an interface compiler which, given an interface grammar for a component, automatically generates a *stub* for that component. This stub acts a parser for incoming call sequences; it checks that the calls conform to the grammar and generates return values according to that grammar. Moreover, the same grammar can be used to create a *driver* that generates call sequences and checks that the values returned by the component conform to it. The compiler

was applied to perform both client and server side verification on two real-world services, including PayPal's Express Checkout, and allowed us to discover a number of mismatches between the implementation of the services and their documentation. In addition to being feasible and efficient, our approach differs from related work mentioned in Section 5 by enabling us to validate and test properties where control flow and message content cannot be handled separately.

2 Web Service Interface Contracts

An interface contract is a set of conventions and constraints that must be fulfilled to ensure a successful interaction with a given web service. Elicitation and enforcement of such contracts has long been advocated [20], and interface documents such as WSDL provide a basic form of specification for syntactical requirements on SOAP messages and request-response patterns. Although many web services are composed of such simple request-response patterns of independent operations, in practice a fair number of services also exhibit long-running behavior that spans multiple requests and responses. This is especially true of commerce-related web services, where concepts such as "purchase transactions" and "shopping carts" naturally entail some form of multi-step operations.

2.1 The PayPal Express Checkout API

A commercial web service suite provided by the PayPal company, called the PayPal Web Service API, is an example of a service that supports multi-step operations. Through its web site, PayPal allows to transfer money to and from credit card and bank accounts between its registered members or other financial institutions. In addition to direct usage by individuals, an organization wishing to use these functionalities from its own web site can do so through PayPal's web service API. All transactions can be processed in the background between the organization and PayPal by exchanging SOAP messages that replace the standard access to PayPal's portal.

PayPal's API is public and its documentation can be freely accessed [1]. The sum of all constraints, warnings, side notes and message schemas found in this documentation constitutes the actual interface contract to the web service API. We shall see that this contract is subject to data and control-flow constraints, and that these constraints can be formally specified using interface grammars.

To illustrate our point, we concentrate on a subset of PayPal's API called "Express Checkout", which allows for a simplified billing and payment between an organization and a customer. The organization simply sends PayPal a total amount to be charged to the customer; PayPal then performs the necessary background checks and confirmations with the customer, after which the organization retrieves a transaction number which can be used to execute the money transfer.

Express Checkout is performed in three steps, each corresponding to a request-response pattern of XML messages. The first step is to create an Express Checkout instance through the SetExpressCheckout message, whose structure, defined

```
<PaymentDetails>
  <Token>1234</Token>
  <OrderTotal>50</OrderTotal>
  <PaymentDetailsItems>
    <PaymentDetailsItem>
      <Name>...</Name>
      <Number>...</Number>
      <Quantity>...</Quantity>
      <Amount>...</Amount>
    </PaymentDetailsItem>
    ...
  </PaymentDetailsItems>
  <PaymentAction>Sale</PaymentAction>
</PaymentDetails>
```

(a) SetExpressCheckoutRequest

```
<Token>...</Token>
<PayerID>...</PayerID>
<PaymentDetailsItems>
...
</PaymentDetailsItems>
```

(b) GetExpressCheckoutDetails

```
<Token>...</Token>
<PayerID>...</PayerID>
<PaymentDetailsItems>
...
</PaymentDetailsItems>
<PaymentAction>Sale</PaymentAction>
```

(c) DoExpressCheckoutPaymentRequest

```
<Token>...</Token>
<PaymentInfo>
  <TransactionID>...<TransactionID>
  <GrossAmount>...<GrossAmount>
  <PendingReason>...<PendingReason>
</PaymentInfo>
```

(d) DoExpressCheckoutPaymentResponse

Fig. 2: Request and response messages from PayPal's Express Checkout API

in the WSDL specification, is shown in Figure 2a. This message provides a total for the order, as well as (optionally) a list of items intended to detail the contents of the order the client is billed for. PayPal's response to this message consists of a single Token element, whose value will be used in subsequent messages to refer to this particular instance of Express Checkout. The PaymentAction element (Figure 2c) can take the value "Sale", indicating that this is a final sale, or "Authorization" and "Order" values indicating that this payment is either a basic or an order authorization, respectively.

The second step consists of obtaining additional details on the Express Checkout through the GetExpressCheckoutDetails operation. The request message simply requires a token identifying an Express Checkout instance; the response to this message is structured as in Figure 2b. It repeats the payment details and token fields from the previous request, and adds a PayerID element. This element is then used in the last operation, DoExpressCheckoutPayment (Figure 2c); the response to this message (Figure 2d) completes the Express Checkout procedure.

2.2 Interface Grammars for Web Services

Interface grammars were proposed as a new language for the specification of component interfaces [15, 16]. The core of an interface grammar is a set of production rules that specifies all acceptable method call sequences for the given component. An interface grammar is expressed as a series of productions of the form $a(v_1, \ldots, v_n) \rightarrow A$. The $v_1, \ldots, v_n$ are lexically scoped variable names corresponding to the parameters of the non-terminal a. A is the right hand side of the production, which may contain the following:

$$
\begin{aligned}
start &\rightarrow !\textsc{seco}(\mathrm{doc}_1, \mathrm{items}, \mathrm{token}, \mathrm{action}); ¡\textsc{seco}(\mathrm{doc}_2, \mathrm{token}); \\
&\quad start; details(\mathrm{items}, \mathrm{token}, \mathrm{action}, \mathrm{payerid}); start \\
&\mid \epsilon \\
details(\mathrm{items}, \mathrm{token}, \mathrm{action}, \mathrm{payerid}) &\rightarrow !\textsc{gecod}(\mathrm{doc}_1, \mathrm{token}); ¡\textsc{gecod}(\mathrm{doc}_2, \mathrm{token}, \mathrm{payerid}); \\
&\quad do(\mathrm{items}, \mathrm{token}, \mathrm{action}, \mathrm{payerid}) \\
do(\mathrm{items}, \mathrm{token}, \text{“}Sale\text{”}, \mathrm{payerid}) &\rightarrow !\textsc{decop}(\mathrm{doc}_1, \mathrm{token}, \mathrm{payerid}, \mathrm{items}, \text{“}Sale\text{”}); \\
&\quad ¡\textsc{decop}(\mathrm{doc}_2, \mathrm{token}, \mathrm{transactionid}) \\
do(\mathrm{items}, \mathrm{token}, \mathrm{action}_1, \mathrm{payerid}) &\rightarrow !\textsc{decop}(\mathrm{doc}_1, \mathrm{token}, \mathrm{payerid}, \mathrm{items}, \mathrm{action}_2); \\
&\quad ¡\textsc{decop}(\mathrm{doc}_2, \mathrm{token}, \mathrm{transactionid})
\end{aligned}
$$

Fig. 3: Interface grammar for a PayPal Express Checkout client

- nonterminals, written $nt(v_1, \ldots, v_n)$;
- semantic predicates that must evaluate to true when the production is used during derivation, written $[\![p]\!]$;
- semantic actions that are executed during the derivation, which we express as $\langle\!\langle a \rangle\!\rangle$;
- incoming method calls, written $?m(v_1, \ldots, v_n)$;
- returns from incoming method calls, written $¿m(v_1, \ldots, v_n)$;
- outgoing method calls, written $!m(v_1, \ldots, v_n)$;
- returns from outgoing method calls, written $¡m(v_1, \ldots, v_n)$.

For the purposes of web service verification, the method calls in the interface grammar correspond to the web service operations. For example, the interface grammar shown in Figure 3 represents the client interface for a simplified version of the PayPal service described previously. Terminal symbols SECO, GECOD and DECOP stand respectively for operations SetExpressCheckout, GetExpressCheckoutDetails and DoExpressCheckoutPayment; the ! and ¡ symbols denote the request and response message for each of these operations. The ? and ¿ symbols, which are not used in our example, would indicate that the server, instead of the client, initiates a request-response pattern.

Nonterminal symbols in interface grammars are allowed to have parameters [15]. The "doc_i" symbol in each message refers to the actual XML document corresponding to that particular request or response; it is assumed fresh in all of its occurrences. Remaining parameters enable us to propagate the data values from that document that might be used as arguments of the web service operations. Because we need to be able to pass data to the production rules as well as retrieve them, we use call-by-value-return semantics for parameters.

By perusing PayPal's API documentation, it is possible to manually define the simple interface grammar shown in Figure 3, which captures a number of important requirements on the use of the PayPal API:

1. Multiple Set, Get and Do operations for different tokens can be interleaved, but, for each token, the Set, Get and Do operations must be performed in order.

2. The PayerID field in the DoExpressCheckoutPaymentRequest must be the one returned by the GetExpressCheckoutDetails response with matching Token element.
3. If the action element of the SetExpressCheckout operation is set to "Sale", it cannot be changed in the DoExpressCheckoutPayment; otherwise, the Get and Do operations can have different action values.
4. To ensure that every Express Checkout instance is eventually complete, every SetExpressCheckout operation must be matched to subsequent GetExpressCheckoutDetails and DoExpressCheckoutPayment requests.

Although all these constraints are mentioned in the service's documentation in some form or another, none of them can be formally described through the WSDL interface document.

3 Translating WSDL to Interface Grammars

While interface grammars can express complex interfaces that involve both data-flow and control-flow constraints, writing such grammars manually requires a surprisingly large amount of boilerplate code. Crafting the appropriate data structures, verifying the result and extracting the data, even for one operation, requires as much code as the entire interface grammar. Moreover, parameters such as "items" and "token" refer to actual elements inside "doc", but the grammar in Figure 3 offers no way of actually specifying how the document and its parts are structured or related. To alleviate this difficulty, we developed a tool that uses type information to automatically translate the data structures associated with a WSDL specification into an interface grammar, without user input.

3.1 Translation from XML Schema to Interface Grammars

A WSDL specification is a list of exposed operations along with the type of the parameters and return values. It encodes all types using XML Schema. Since XML Schema itself is verbose, we use the *Model Schema Language (MSL)* formalism [10], which encodes XML Schema in a more compact form. More precisely, we define a simplified version of MSL that handles all the portions of XML Schema we found necessary in our case studies:

$$g \to b \mid t[g_0] \mid g_1\{m,n\} \mid g_1,\ldots,g_k \mid g_1|\ldots|g_k \tag{1}$$

Here $g, g_0, g_1, \ldots, g_k$ are all MSL types; b is a basic data type such as Boolean, integer, or string; t is a tag; and m and n are natural numbers such that $m < n$ (n may also be ∞).

The MSL type expressions are interpreted as follows: $g \to b$ specifies a basic type b; $g \to t[g_0]$ specifies the sub-element t of g, whose contents are described by the type expression g_0; $g \to g_1\{m,n\}$, where $n \neq \infty$, specifies an array of g_1s with at least m elements and at most n elements; $g \to g_1\{m,\infty\}$ specifies

an unbounded array of g_1s with at least m elements; $g \rightarrow g_1, \ldots, g_k$ specifies an ordered sequence, with each of the g_is listed one after the other; and $g \rightarrow g_1 | \ldots | g_k$ specifies choice, where g is one of the g_is. We denote the language of type expressions generated by Equation (1) to be $\mathcal{XML}$.

For example, the type for the DoExpressCheckoutPaymentResponse message (Figure 2c) is the following:

Token[**string**], **PaymentInfo**[
TransactionID[**string**], **GrossAmount**[**int**], **PendingReason**[**string**]]

As a more complex example, the SetExpressCheckoutRequest message (Figure 2a) is of the following type:

Token[**string**]$\{0,1\}$,
PaymentDetails[
OrderTotal[**int**],
PaymentDetailsItems[
PaymentDetailsItem[
Name[**string**]$\{0,1\}$, **Number**[**string**]$\{0,1\}$, **Quantity**[**int**]$\{0,1\}$,
Amount[**int**]$\{0,1\}$,
]$\{1,\infty\}$
]$\{0,\infty\}$,
PaymentAction[**string**]$\{0,1\}$]

These type expressions can be used to generate XML documents. However, to communicate with a SOAP server, we chose to use Apache Axis, a library that serializes Java objects into XML. Accordingly, we create Java objects from XML type expressions, and do so in the same way that Axis maps WSDL to Java objects.

XML Schema and the Java type system are very different and, hence, mapping from one to the other is not trivial. However, since such a mapping is already provided by Axis, all we have to do is the follow the same mapping that Axis uses:

1. $g \rightarrow b$ is mapped to a Java basic type when possible (for example, with Booleans or strings). Because XML Schema integers are unbounded and Java integers are not, we must use a specialized Java object rather than native integers.
2. $g \rightarrow t[g_0]$ is mapped to a new Java class whose name is the concatenation of the current name and t; this class contains the data in g_0, and will be set to the t field in the current object.
3. $g \rightarrow g_1\{0,1\}$ is mapped to either **null** or the type mapped by g_1.
4. $g \rightarrow g_1\{m,n\}$ is mapped to a Java array of the type mapped by g_1.
5. $g \rightarrow g_1, \ldots, g_k$ is mapped to a new Java class that contains each of the g_is as fields.
6. $g \rightarrow g_1 | \ldots | g_k$ is mapped to a new Java interface that each of the g_is must implement.

The rules for the WSDL to interface grammar translation are shown in Figure 4. The translation is defined by the function $\mathbf{p}$, which uses the auxiliary functions $\mathbf{r}$ (which gives unique names for type expressions suitable for use in grammar nonterminals) and $\mathbf{t}$ (which gives the name of the new Java class created in the Axis mapping of $g \to t[g_0]$). By applying $\mathbf{p}[\![g]\!]$ to an XML Schema type expression g, we compute several grammar rules to create Java object graphs for all possible instances of the type expression g. The start symbol for the generated interface grammar is $\mathbf{r}[\![g]\!]$.

$$\begin{aligned}
\mathbf{p} &: \mathcal{XML} \to \mathbf{Prod} \\
\mathbf{r} &: \mathcal{XML} \to \mathbf{NT} \\
\mathbf{t} &: \mathcal{XML} \to \mathbf{Type}
\end{aligned}$$

$$\mathbf{p}[\![g \quad = \mathbf{boolean}]\!] = \left\{ \begin{array}{l} \mathbf{r}[\![g]\!](x) \to \langle\!\langle x = \mathbf{true} \rangle\!\rangle, \\ \mathbf{r}[\![g]\!](x) \to \langle\!\langle x = \mathbf{false} \rangle\!\rangle \end{array} \right\} \tag{2}$$

$$\mathbf{p}[\![g \quad = \mathbf{int}]\!] = \left\{ \begin{array}{l} \mathbf{r}[\![g]\!](x) \to \langle\!\langle x = \mathbf{0} \rangle\!\rangle, \\ \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g]\!](x); \langle\!\langle x = x + 1 \rangle\!\rangle \end{array} \right\} \tag{3}$$

$$\mathbf{p}[\![g \quad = \mathbf{string}]\!] = \left\{ \begin{array}{l} \mathbf{r}[\![g]\!](x) \to \langle\!\langle x = \texttt{""} \rangle\!\rangle, \\ \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g]\!](x); \langle\!\langle x = x \| c \rangle\!\rangle \quad \text{for every } c \end{array} \right\} \tag{4}$$

$$\mathbf{p}[\![g = \{c_1, \ldots, c_n\}]\!] = \left\{ \mathbf{r}[\![g]\!](x) \to \langle\!\langle x = \texttt{"}c_i\texttt{"} \rangle\!\rangle \quad \text{for every } c_i \right\} \tag{5}$$

$$\mathbf{p}[\![g \quad = t[g']]\!] = \left\{ \begin{array}{l} \mathbf{r}[\![g]\!](x) \to \langle\!\langle \text{if } (x \equiv \mathbf{null})\, x = \mathbf{new}\, \mathbf{t}[\![g]\!] \rangle\!\rangle; \\ \qquad\qquad \mathbf{r}[\![g']\!](y); \langle\!\langle x.t = y \rangle\!\rangle \end{array} \right\} \cup \mathbf{p}[\![g']\!] \tag{6}$$

$$\mathbf{p}[\![g \quad = g'\{0,1\}]\!] = \left\{ \begin{array}{l} \mathbf{r}[\![g]\!](x) \to \langle\!\langle x = \mathbf{null} \rangle\!\rangle, \\ \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g']\!](x), \end{array} \right\} \cup \mathbf{p}[\![g']\!] \tag{7}$$

$$\mathbf{p}[\![g \quad = g'\{0,\infty\}]\!] = \left\{ \begin{array}{l} \mathbf{r}[\![g]\!](x) \to \langle\!\langle x = [] \rangle\!\rangle, \\ \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g']\!](y); \mathbf{r}[\![g]\!](x); \langle\!\langle x = x \| y \rangle\!\rangle \end{array} \right\} \cup \mathbf{p}[\![g']\!] \tag{8}$$

$$\mathbf{p}[\![g \quad = g'\{0,n\}]\!] = \left\{ \begin{array}{lcl} \mathbf{r}[\![g]\!](x) & \to & \langle\!\langle x = [] \rangle\!\rangle, \\ \mathbf{r}[\![g]\!](x) & \to & \mathbf{r}[\![g']\!](y); \langle\!\langle x = [y] \rangle\!\rangle, \\ & \cdots & \\ \mathbf{r}[\![g]\!](x) & \to & \mathbf{r}[\![g']\!](y_1); \ldots; \mathbf{r}[\![g']\!](y_n); \\ & & \langle\!\langle x = [y_1, \ldots, y_n] \rangle\!\rangle \end{array} \right\} \cup \mathbf{p}[\![g']\!] \tag{9}$$

$$\begin{aligned} \mathbf{p}[\![g \quad = g'\{m,n\}]\!] &= \left\{ \begin{array}{l} \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g']\!](y_1); \ldots; \mathbf{r}[\![g']\!](y_m); \\ \qquad\qquad \mathbf{r}[\![g'']\!](x); \langle\!\langle x = [y_1, \ldots, y_m] \| x \rangle\!\rangle \end{array} \right\} \\ &\quad \cup \mathbf{p}[\![g'']\!] \cup \mathbf{p}[\![g']\!] \text{ where } g'' \to g'\{0, n-m\} \end{aligned} \tag{10}$$

$$\mathbf{p}[\![g \quad = g_1, \ldots, g_k]\!] = \{\mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g_1]\!](x); \ldots; \mathbf{r}[\![g_k]\!](x)\} \cup \bigcup_{i=1}^{k} \mathbf{p}[\![g_i]\!] \tag{11}$$

$$\mathbf{p}[\![g \quad = g_1 | \ldots | g_k]\!] = \bigcup_{i=1}^{k} \{\mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g_i]\!](x)\} \cup \mathbf{p}[\![g_i]\!] \tag{12}$$

For a nonterminal g, $\mathbf{p}[\![g]\!]$ is the set of associated grammar rules, $\mathbf{r}[\![g]\!]$ is a unique name suitable for a grammar nonterminal, $\mathbf{t}[\![g]\!]$ is the unique Java type for that position in the XML Schema grammar, and x and y designate an XML document or subdocument.

Fig. 4: MSL to interface grammar translation rules

Rule (2) translates Boolean types by simply enumerating both possible values. Calling $\mathbf{r}[\![g]\!](x)$ with an uninitialized variable x will set x to either **true** or

false. Rule (3) translates integer numbers to a Java instance by starting at 0 and executing an unbounded number of successor operations. If the number is bounded we can generate it more efficiently by creating one production for each value. However, Rule (3) allows us to generate an infinite number of values. We can also accommodate negative integers using Rule (3) and then choosing a sign.

Rule (4) translates strings to Java strings. It starts with an empty string and concatenates an unbounded number of characters onto it, to generate all possible string values. It should be noted that strings are frequently used as unspecified enumerations, have possible correlations with other parts of the object graph, or have some associated structure they should maintain (as in search queries), etc. Accordingly, the automatically generated grammar can be refined to something more restricted but also more useful by manually changing these rules.

Rule (5) takes care of enumerated types by providing one rule to generate each possible value of that type.

Rule (6) translates tags into Java objects. The rule is simple; we figure out which Java type Axis is using for this position using $\mathbf{t}[\![g]\!]$, if it is not already initialized (which can happen if we are applying Rule (11)) instantiate it, recursively process its contents, and then set the contents to the t field on the object we are currently working on. Rule (7) translates optional elements into Java objects by having two rules, one for **null** and the other to generate the object.

Rule (8) translates unbounded arrays into Java objects. We start with the base case of an empty array and concatenate objects onto it. Rule (9) translates bounded arrays into Java objects, by simply generating n rules, one for each potential object. Although we give this simple rule here for readability, in our implementation we handle this case more efficiently.

Rule (10) translates general arrays, that may have a minimum number of objects greater than 0, to a situation where one of Rule (8) or Rule (9) applies. Rule (11) translates sequences into Java objects; we simply apply each of the sub-rules to the object graph under examination in sequence. Rule (12) translates alternations into Java objects; we pick one of the sub-rules and apply it.

As an example of translation, consider the MSL type for the DoExpressCheckoutPaymentResponse message in the PayPal WSDL specification mentioned above. First, the production rules for the basic types string and integer are:

$$
\begin{aligned}
&\mathit{string}(\mathrm{doc}) \rightarrow \langle\!\langle \mathrm{doc} = \texttt{""} \rangle\!\rangle \\
&\mathit{string}(\mathrm{doc}) \rightarrow \mathit{string}(\mathrm{doc}); \langle\!\langle \mathrm{doc} = \mathrm{doc} \| c \rangle\!\rangle \quad \text{for every character } c \\
&\mathit{int}(\mathrm{doc}) \rightarrow \langle\!\langle \mathrm{doc} = \mathbf{0} \rangle\!\rangle \\
&\mathit{int}(\mathrm{doc}) \rightarrow \mathit{int}(\mathrm{doc}); \langle\!\langle \mathrm{doc} = \mathrm{doc} + 1 \rangle\!\rangle
\end{aligned}
$$

The message type consists of a sequence. For the first element of the sequence, we need to apply Rule (6) followed by Rule (4), resulting in the following grammar production:

$$a(\text{doc}) \rightarrow \langle\!\langle \text{if } (\text{doc} \equiv \textbf{null})\, \text{doc} = \textbf{new } \texttt{Token} \rangle\!\rangle; \mathit{string}(\text{doc}_1); \langle\!\langle \text{doc}.\textbf{Token} = \text{doc}_1 \rangle\!\rangle$$

with start symbol a. Applying these productions can assign to *doc* a subdocument like <Token>abc</Token>. Nonterminal a is responsible for the creation of the Token element, and repeated application of the productions for the *string* nonterminal creates an arbitrary value for the string field.

For the second element of the sequence we apply Rule (6) which leads to another sequence. Then we apply Rule (11) followed by three applications of Rule (6), two applications of Rule (4) and one application of Rule (3). The resulting productions are:

$$\begin{aligned}
b(\text{doc}) \rightarrow\ & \langle\!\langle \text{if } (\text{doc} \equiv \textbf{null})\, \text{doc} = \textbf{new } \texttt{PaymentInfo} \rangle\!\rangle; c(\text{doc}_1); d(\text{doc}_1); e(\text{doc}_1); \\
& \langle\!\langle \text{doc}.\textbf{PaymentInfo} = \text{doc}_1 \rangle\!\rangle \\
c(\text{doc}) \rightarrow\ & \langle\!\langle \text{if } (\text{doc} \equiv \textbf{null})\, \text{doc} = \textbf{new } \texttt{PaymentInfoTransactionID} \rangle\!\rangle; \mathit{string}(\text{doc}_1); \\
& \langle\!\langle \text{doc}.\textbf{PaymentInfoTransactionID} = \text{doc}_1 \rangle\!\rangle \\
d(\text{doc}) \rightarrow\ & \langle\!\langle \text{if } (\text{doc} \equiv \textbf{null})\, \text{doc} = \textbf{new } \texttt{PaymentInfoGrossAmount} \rangle\!\rangle; \mathit{int}(\text{doc}_1); \\
& \langle\!\langle \text{doc}.\textbf{PaymentInfoGrossAmount} = \text{doc}_1 \rangle\!\rangle \\
e(\text{doc}) \rightarrow\ & \langle\!\langle \text{if } (\text{doc} \equiv \textbf{null})\, \text{doc} = \textbf{new } \texttt{PaymentInfoPendingReason} \rangle\!\rangle; \mathit{string}(\text{doc}_1); \\
& \langle\!\langle \text{doc}.\textbf{PaymentInfoPendingReason} = \text{doc}_1 \rangle\!\rangle
\end{aligned}$$

with start symbol b. Finally, we apply Rule (11) one more time resulting in one additional nonterminal and production:

$$\mathit{DoExpressCheckoutPaymentResponse}(\text{doc}) \rightarrow a(\text{doc}); b(\text{doc})$$

3.2 Control-Flow and Messages

Using the translation scheme described above, terminal symbols standing for messages in the grammar of Figure 3 can be expanded into productions for validating or generating individual message instances. For example, the ¡DECOP terminal symbol refers to a message of type DoExpressCheckoutPaymentResponse. Generating such a message simply amounts to expanding the respective message productions according to the derivation rules we have just shown.

Recall that production symbols in an interface grammar can carry additional parameters that can be used to refer to specific elements of messages. These parameters can be passed on from message to message to express correlations between parameters across a whole transaction.

In the case of the ¡DECOP symbol, we attach two parameters: token and transactionid, standing for the values of message elements of the same name. We must therefore associate these two variables with the actual content of the production that relates to these values:

$$
\begin{aligned}
a'(\mathrm{doc}, \mathrm{token}) \rightarrow\ & \langle\!\langle \text{if } (\mathrm{doc} \equiv \textbf{null})\, \mathrm{doc} = \textbf{new } \texttt{Token} \rangle\!\rangle; \mathit{string}(\mathrm{token}); \\
& \langle\!\langle \mathrm{doc}.\textbf{Token} = \mathit{token} \rangle\!\rangle \\
b'(\mathrm{doc}, \mathrm{transactionid}) \rightarrow\ & \langle\!\langle \text{if } (\mathrm{doc} \equiv \textbf{null})\, \mathrm{doc} = \textbf{new } \texttt{PaymentInfo} \rangle\!\rangle; \\
& c'(\mathrm{doc}_1, \mathrm{transactionid}); d(\mathrm{doc}_1); e(\mathrm{doc}_1); \\
& \langle\!\langle \mathrm{doc}.\textbf{PaymentInfo} = \mathrm{doc}_1 \rangle\!\rangle \\
c'(\mathrm{doc}, \mathrm{transactionid}) \rightarrow\ & \langle\!\langle \text{if } (\mathrm{doc} \equiv \textbf{null})\, \mathrm{doc} = \textbf{new } \texttt{PaymentInfoTransactionID} \rangle\!\rangle; \\
& \mathit{string}(\mathrm{transactionid}); \\
& \langle\!\langle \mathrm{doc}.\textbf{PaymentInfoTransactionID} = \mathrm{transactionid} \rangle\!\rangle
\end{aligned}
$$

Finally, the rule for ¡DECOP itself can be obtained by:

$$
\text{¡DECOP}(\mathrm{doc}, \mathrm{token}, \mathrm{transactionid}) \rightarrow a'(\mathrm{doc}, \mathrm{token}); b'(\mathrm{doc}, \mathrm{transactionid});
$$

This mechanism is not restricted to primitive types; for example, the "items" argument of the !SECO message stands for the list of items; this element itself is formed of multiple item elements with values for name, amount, and so on.

This particular characteristic of our translation to interface grammars is fundamental. By expressing message structures, parameter values and control flow in a uniform notation, all such properties of a given service are taken into account in one specification framework. For example, by using the above rules to simulate an Express Checkout client, we have that: 1) if the client invokes SetExpressCheckout with some token i, then the client expects a response with the same token value; 2) the client is guaranteed to eventually invoke Get and Do with that same token i. Additionally, if the client invokes SetExpressCheckout with an action value of "Sale" for token i, then the DoExpressCheckoutPayment message that will be eventually sent for token i will also have the value "Sale". These constraints could not be handled if the messages were generated by a procedure independent of the control flow constraints.

4 Experiments

To demonstrate the value of our approach, we studied two web services: the Amazon E-Commerce Service provided by Amazon.com and the PayPal Web Service API that we used as a running example throughout the paper.

4.1 Amazon E-Commerce Service

The Amazon E-Commerce Service (AWS-ECS) [3] provides access to Amazon's product data through a SOAP interface specified with WSDL. It was analyzed in an earlier paper [17]; however, although it was not mentioned at the time, the interface grammar for the six key operations (ItemSearch, CartCreate, CartAdd, CartModify, CartGet, and CartClear) was generated automatically from

the WSDL specification of the AWS-ECS. These six operations also have several control flow constraints that are not stated in the WSDL specification of the AWS-ECS. We extended the automatically generated interface grammar by adding these extra constraints. The data summarized below, and the interface grammar itself, are described in more detail in [17].

We used the interface grammar both for client and server side verification, as shown in Figure 1. The AWS-ECS client we used in our experiments is called the AWS Java Sample. This client performs no validation on its input data whatsoever. It is intended as a programming example showing how to use the SOAP and REST interfaces, not as something to use. Hence, it serves as a suitable vehicle to demonstrate the bug finding capabilities of our approach.

We fed the interface grammar for the AWS-ECS to our interface compiler and generated a service stub for the AWS-ECS. We combined this service stub with the AWS Java Sample for client verification. We used the Java PathFinder (JPF) [9] to systematically search the state space of the resulting system. Note that a model checker like JPF is not able to analyze the AWS Java Sample without the automatically generated service stub provided by our interface compiler.

We analyzed three types of errors that the client, were it doing proper input validation, would catch: type failures happen when the user enters a string when an integer is expected; data failures occur when the user attempts to add a nonexistent item to a nonexistent cart (the request is syntactically valid, but nonsensical); uncorrelated data failures involve two operations that are in the correct sequence, but the data associated with the two calls violates the extra constraints (for example, editing an item that was previously removed from the cart). We were able to discover the type failures in 12.5 seconds using 25 MB of memory, the data failures in 11.1 seconds using 25 MB of memory and the uncorrelated data failures in 20.8 seconds using 43 MB of memory.

For server verification, our interface compiler takes the interface specification as input and automatically generates a driver that sends SOAP requests to the web service. We ran ten tests using a sentence generator that chooses the next production randomly. In each of these tests, the sentence generator was run until it produced 100 SOAP message sequences, which were sent to the AWS-ECS server. The average execution time for the tests was 430.2 seconds (i.e., 4.3 seconds per sequence). On average, the driver took 17.5 steps per derivation, and each such derivation produced 3.2 SOAP requests.

These tests uncovered two errors, corresponding to mismatches between the interface grammar specification and the AWS-ECS implementation:

1. The AWS-ECS implementation does not allow multiple add requests for the same item, although this is not clear from the specification of the service.
2. We assumed that a shopping cart with no items in it would have an items array with zero length. However, in the implementation this scenario leads to a shopping cart with a null items array. This was not clarified in the AWS-ECS specification.

4.2 PayPal Express Checkout Service

As a second case study, we conducted server side verification for PayPal's Express Checkout API. The running example in earlier sections is a simplified version of this API. As we did for the server side verification of the AWS-ECS service, we used a random sentence generator algorithm that sends SOAP requests to PayPal's web service. Our tests uncovered two errors. Again, these errors correspond to discrepancies between the interface grammar specification and the API's actual implementation:

1. In a SetExpressCheckout request, elements CancelURL and ReturnURL cannot be arbitrary strings; they must be valid URLs. This is not written in the API documentation or in the WSDL, which only specify it must be a string. It took 5.7 seconds to find this error.
2. The implementation does not allow a client to set its own token in a SetExpressCheckout request. If the client does not use a token previously returned by another SetExpressCheckout request, it has to set the token to the empty string and reuse what SetExpressCheckout gives back. Again this constraint was not clear from the documentation. It took 2.5 seconds to catch this error.

Once we modified the interface grammar specification to reflect these constraints, the driver did not produce any more errors. The round-trip time to generate each new message from the grammar, send it, get and parse the response from PayPal took about 1 second.

5 Related Work

Earlier work has been done on grammar-based testing. For example, Sirer and Bershad [21] have developed a grammar-based test tool, *lava*, with a focus on validating Java Virtual Machine implementations. Test data has been generated using enhanced context-free grammars [19], regular grammars [8] and attributed grammars [12]. None of these tools focus on web service verification —they use grammars to characterize inputs rather than interfaces.

Some approaches attempt to automate the testing of web services by taking advantage of their WSDL definitions. Available tools like soapUI [2] allow a user to create so-called "mock web services" whose goal is to mimic the actual web service requests and responses; for each such operation, the tool generates a message skeleton that the user can then manually populate with data fields. Other works automate this process entirely by simulating a web service through the generation of arbitrary, WSDL-compliant messages when requested [4, 7].

On the other hand, other works attempt to validate incoming and outgoing messages to ensure they are WSDL-compliant. The Java API for XML Web Services (JAX-WS)[1] provides a validator for that purpose; the IBM Web Service

[1] https://jax-ws.dev.java.net/

Validation Tool[2] validates a trace of SOAP messages against WSDL specifications. Cacciagrano et al. [11] push the concept further and validate not only the structure of messages, but also additional constraints such as dependencies between values inside a message.

However, all these previous approaches treat request-response as patterns independently of each other; therefore, they do not allow properties where values generated in some messages constrain the control flow of the web service, as we have shown is the case in PayPal's Express Checkout.

6 Conclusion

We proposed and implemented a translator to automatically generate an interface grammar skeleton from a WSDL specification. This interface grammar skeleton can be combined with control flow constraints to generate an interface specification that characterizes both control and data-flow constraints in a uniform manner. Using the actual documentation and WSDL specification from the PayPal Express Checkout API, we have shown how such automatically generated grammar skeletons can be extended with control flow constraints to obtain interface grammars that specify the interaction behavior of web services. These interface grammars can then be automatically converted to web service stubs and drivers to enable verification and testing. We also applied these techniques to a client for the key interfaces of the Amazon E-Commerce Service and also to the Amazon E-Commerce Service server directly, and have demonstrated that our approach is feasible and efficient.

References

1. PayPal web service API documentation, 2008. http://www.paypal.com.
2. soapUI: the web services testing tool, 2009. http://www.soapui.org/.
3. Amazon web services. http://solutions.amazonwebservices.com/.
4. X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. WSDL-based automatic test case generation for web services testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE) 2005*, pages 207–212, 2005.
5. F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006)*, pages 63–71, 2006.
6. L. Baresi, S. Guinea, R. Kazhamiakin, and M. Pistore. An integrated approach for the run-time monitoring of BPEL orchestrations. In *Proceedings of the First European Conference Towards a Service-Based Internet (ServiceWave 2008)*, pages 1–12, 2008.
7. C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. Towards automated WSDL-based testing of web services. In A. Bouguettaya, I. Krüger, and T. Margaria, editors, *ICSOC*, volume 5364 of *Lecture Notes in Computer Science*, pages 524–529, 2008.

[2] http://www.alphaworks.ibm.com/tech/wsvt

8. J. A. Bauer and A. B. Finger. Test plan generation using formal grammars. In *Proceedings of the 4th International Conference on Software Engineering*, pages 425–432, Munich, Germany, September 1979.
9. G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder: Second generation of a Java model checker. In *Proceedings Workshop on Advances in Verification*, 2000.
10. A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL: a model for W3C XML Schema. In *Proceedings of the 10th International World Wide Web Conference*, pages 191–200, 2001.
11. D. Cacciagrano, F. Corradini, R. Culmone, and L. Vito. Dynamic constraint-based invocation of web services. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 138–147. Springer, 2006.
12. A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proceedings of the 5th International Conference on Software Engineering*, pages 170–178, New York, NY, USA, March 1981.
13. S. Hallé and R. Villemaire. Runtime monitoring of message-based workflows with data. In *Proceedings of the 12th International Enterprise Distributed Object Computing Conference (EDOC 2008)*, pages 63–72, 2008.
14. S. Hallé and R. Villemaire. Browser-based enforcement of interface contracts in web applications with BeepBeep. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 648–653. Springer, 2009.
15. G. Hughes and T. Bultan. Extended interface grammars for automated stub generation. In *Proceedings of the Automated Formal Methods Workshop (AFM 2007)*, 2007.
16. G. Hughes and T. Bultan. Interface grammars for modular software model checking. *IEEE Trans. Software Eng.*, 34(5):614–632, 2008.
17. G. Hughes, T. Bultan, and M. Alkhalaf. Client and server verification for web services using interface grammars. In T. Bultan and T. Xie, editors, *TAV-WEB*, pages 40–46. ACM, 2008.
18. K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005)*, pages 257–265, 2005.
19. P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.
20. G. Meredith and S. Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.
21. E. Sirer and B. N. Bershad. Using production grammars in software testing. In *Proceedings of DSL'99: the 2nd Conference on Domain-Specific Languages*, pages 1–13, Austin, TX, US, 1999.