Client and Server Verification for Web Services Using Interface Grammars*

Graham Hughes Tevfik Bultan Muath Alkhalaf
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
{graham,bultan,muath}@cs.ucsb.edu

ABSTRACT

Web services provide a promising framework for developing interoperable software components that interact with each other across organizational boundaries. For this framework to be successful, the client and the server for a service have to interact with each other based on the published service interface specification. If either the client or the server deviate from the interface specification, the client-server interaction will lead to errors. We present a framework for checking interface conformance for web services. Given an interface specification, we automatically generate web service server stubs (for client verification) and drivers (for server verification) and then use these stubs and drivers to check the conformance of the client and server to the interface specification. We implemented this framework by using interface grammars as the interface specification language. We developed an interface compiler that automatically generates a stub or a driver from a given interface grammar. We conducted a case study by applying these techniques to the Amazon E-Commerce Service.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

interface grammars, web services

1. INTRODUCTION

By providing a framework that enables web accessible software applications to interact with each other through the Internet, Web services provide a promising next step in the evolution of electronic commerce. A crucial attribute of the web services framework is interoperability. In this framework, different software components written by different organizations should be able to interact with each other based on the available interface specifications. Hence, conformance to the interface specification becomes very important. If either the client or the server deviate from the interface specification, the client-server interaction will lead to errors. Note that it may not be easy to test the client and server together since they may not belong to the same organization.

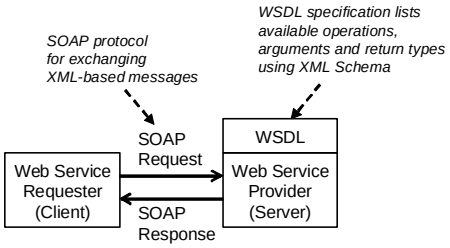

Figure 1: Basic architecture for web services

In this paper we present a framework that addresses this problem. Our basic idea is, given an interface specification, to automatically generate web service stubs (for client verification) and drivers (for server verification) and then use these stubs and drivers to check the conformance of the client and the server to the interface specification. We implemented this framework by using interface grammars as the interface specification language. We developed a compiler that automatically generates stubs and drivers from a given interface grammar.

Web services interact with each other by exchanging messages encoded using the eXtensible Markup Language (XML) [21]. XML Schema [22] provides a type system for XML messages and the Simple Object Access Protocol (SOAP) [18] is 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 format to invoke the service, using the Web Services Description Language (WSDL) [20]. 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 SOAP protocol as seen in Figure 1. The WSDL specification serves as the contract between the client and the server that defines the valid interactions.

Our web service verification framework is based on an interface compiler that automatically creates stubs and drivers for client and server verification. The input language for our interface compiler is interface grammars. Interface grammars are expressive enough

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

TAV-WEB – Workshop on Testing, Analysis and Verification of Web Software, July 21, 2008

to specify both control and data constraints that arise in web service interfaces. Given server-side and client-side interface grammar specifications, we use our interface compiler to generate a server stub and a service driver. The server stub is a top-down parser which uses the incoming SOAP requests as a lookahead. The service driver, on the other hand, is a language generator, that generates SOAP request sequences based on the interface grammar.

Related Work: There has been earlier work on grammar-based testing. Sirer and Bershad [17] have developed a grammar-based test tool *lava,* with a focus on validating Java Virtual Machine implementations. Maurer [14, 15] generates test data with an enhanced context free grammar for his DGL tool, Bauer and Finger [4] generate test cases using a regular grammar, and Duncan and Hutchison [10] use attributed grammars to generate test cases. None of these tools focus on web service verification and they use grammars to characterize inputs rather than interfaces.

The use of finite state machines for specification, verification and extraction of interfaces have been studied extensively [9, 8, 19, 2, 6, 5]. Finite state machines are not as expressive as interface grammars and cannot specify nested structures.

In our earlier work, we proposed an approach for modeling interfaces of software components using grammars [12] and extended it with support for modeling recursive data structures [11]. The current paper extends these earlier results in several significant ways. Specifically, our contributions in this paper can be summarized as follows: 1) The framework and tools proposed in this paper enable both client and server verification for web services whereas our earlier work [12] handles only client side verification. 2) This paper focuses on web services and the presented tools produce stubs and drivers that can simulate SOAP calls. 3) We implement and experiment with two sentence generation heuristics for server verification. We report coverage criteria such as production and non-terminal coverage. 4) We conduct a case study and apply our framework to both client and server verification of Amazon Web Services.

The rest of the paper is organized as follows. In Section 2 we describe an Amazon Web Service that we use as a case study. In Section 3 we give an overview of interface grammars and show a sample interface grammar for a simplified version of the Amazon Web Service that we used in our case study. In Section 4 we discuss our interface compiler. In Section 5 we discuss client verification using interface grammars. In Section 6 we discuss server verification using interface grammars. In Section 7 we conclude the paper.

2. AMAZON WEB SERVICE

In order to demonstrate the effectiveness of the approach we propose for verification and testing of web service clients and servers, we have conducted a case study using the Amazon Web Services (AWS) provided by Amazon.com [3]. AWS is a large framework including various services such as Amazon Elastic Compute Cloud (Amazon EC2) for cloud computing, Amazon Flexible Payments Service (Amazon FPS) for financial transactions, which require payment. The Amazon Associates Web Service (also known as the Amazon E-Commerce Service or "ECS"), on the other hand, is a free service that exposes Amazon's product data with the goal of driving traffic back to Amazon's web sites or sales of Amazon products and services. We chose this service as the target of our case study.

The Amazon E-Commerce Service (which we will refer to as AWS-ECS) provides access to Amazon's product data through a SOAP interface specified with WSDL. The scope of the full AWS-ECS is enormous since it provides information about the wide variety of goods Amazon.com sells. The WSDL specification for the AWS-ECS lists 40 operations, almost all of which are differing ways of searching Amazon's product database. We focus on what we believe to be the core of the AWS-ECS API, consisting of the ItemSearch, CartCreate, CartAdd, CartModify, CartGet, and CartClear operations. (Notably absent is any manner for automatically purchasing items in a cart. Amazon prefers that, after the user finds what he/she is looking for, the AWS-ECS clients direct the user to an Amazon web page for processing purchasing transactions.) We informally define the semantics of these operations as follows:

- ItemSearch searches Amazon's database for items matching some set of keyword parameters. 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 cart that represents a request for n copies of that item. All AWS-ECS operations require absolute quantities; that is, it is impossible to say 'add one more of this to the cart'; you must first discover the contents of the cart and then send a message updating them accordingly.
- CartAdd takes a cart, an ASIN and a positive integer n and adds a new row to the cart requesting n copies of that item. It returns the modified cart and a unique item identifier for the new row in that cart. It is illegal to add an ASIN to a cart that already has a row for that ASIN.
- CartModify takes a cart, an item identifier, and a non-negative integer n and alters the row in the cart signified by the item identifier so that it requests instead n copies of the item. The row referred to by the item identifier must exist. If $n = 0$, then the row is deleted entirely. Even though there can only be one row for each ASIN, CartModify only takes the item identifier returned by CartAdd or CartCreate.
- CartGet is a query operations that takes a cart and returns the contents of the cart.
- CartClear takes a cart and removes everything from it, emptying it out. This operation represents one of only two ways to achieve an empty cart; the other is using CartModify with $n = 0$.

The above six operations have several control flow constraints that are not stated in the WSDL specification of the AWS-ECS. Except for ItemSearch and CartCreate, every operation requires that a cart already exists. CartModify can only be called after an item identifier has been retrieved from CartCreate or CartAdd. It is illegal to call CartModify with anything, including previously valid data, after a CartClear. Worse yet, this control flow is data dependent; for example, one CartModify after another on the same item identifier can be okay if the first has an $n > 0$, and fail otherwise.

3. INTERFACE GRAMMARS

In previous work [11, 12], we have proposed *interface grammars* as a new language for the specification of component interfaces. The core of an interface grammar is a set of production rules that specifies all acceptable method call sequences for the given component. Given an interface specification for a component, our interface compiler generates a stub for that component. This stub is a table-driven top-down parser [1] that parses the sequence of incoming method calls (i.e., the method invocations) based on the interface grammar defined by the interface specification.

Our previous work focused on the generation of stubs for Java components (where we define a component as a set of Java classes). Hence, our first interface grammar language only allowed receive events where a receive event represented a method call received by the component stub. In this paper our goal is to verify web services. Since a typical web service involves interaction between a service provider (server) and a client, we can think of two verification tasks: 1) verifying the server by generating a driver that behaves like a client, 2) verifying the client by generating a stub that behaves like the server. In order to support both types of verification, we have extended our interface grammar language to support both receive and send events.

As discussed in [11] we allow nonterminals in interface grammars to have parameters. This enables us to propagate the data values that might be used as arguments of the web service operations. Moreover, it also enables us to specify recursive data structures. 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.

An interface grammar is expressed as a series of productions of the form $a(v_1, \ldots, v_n) \rightarrow A$ where $v_1, \ldots, v_n$ 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 $\langle\!\langle a \rangle\!\rangle$;
- incoming method calls, which we express as $?m(v_1, \ldots, v_n)$;
- returns from incoming method calls, which we express as $¿m(v_1, \ldots, v_n)$;
- outgoing method calls, which we express as $!m(v_1, \ldots, v_n)$;
- returns from outgoing method calls, which we express as $¡m(v_1, \ldots, v_n)$.

Here $v_1, \ldots, v_n$ are variable names, which are lexically scoped.

For the purposes of the web service verification, the method calls in the interface grammar represent the web service operations used. The interface grammar shown in Figure 2 represents the client interface for a simplified version of the AWS-ECS service described in Section 2. To ease presentation, in the interface grammar shown in Figure 2, we consider the following simplified versions of the AWS-ECS operations. In the verification experiments reported in Sections 5 and 6, we used the operations in their full complexity without any simplifications.

- The method SEARCH is a simplification of the AWS-ECS operation ItemSearch. It takes no arguments, and returns the ASIN—the unique identifier for an item in the AWS-ECS database—for some item in the database at random.
- The method CREATE is a simplification of the AWS-ECS operation CartCreate. It takes a ASIN as an argument and returns both a cart containing that ASIN and a unique item identifier for that row in that cart. We omit the numeric argument for the purposes of this discussion, because the only values that affect control flow are 0 and $n > 0$—that is, is the item in the cart or not. Since CartCreate requires a positive integer, any $n > 0$ is sufficient.
- The method ADD is a simplification of the AWS-ECS operation CartAdd. It takes a cart and an ASIN as arguments and returns both the modified cart and a unique item identifier for that row in that cart. Just as with CartAdd, it is illegal to add an ASIN to a cart that already contains it. We ignore here the numerical argument for the same reasons we do for CREATE.
- The method MODIFY is a simplification of the AWS-ECS operation CartModify. It takes a cart, an item identifier and a non-negative integer as arguments and returns the modified cart. Because 0 is a legitimate argument for CartModify, and causes control flow complications if present (as the item referred to is now deleted from the cart) we must respect this distinction in our simplification.
- The method GET is a simplification of the AWS-ECS operation CartGet. It takes a cart and returns it. For our purposes it is a no-op; its utility only becomes apparent when multiple concurrent programs attempt to modify the same cart, a complication we do not address in this simplified example.
- The method CLEAR is a simplification of the AWS-ECS operation CartClear. It takes a cart and returns an empty cart.

We briefly address the semantics of these grammars in Section 4. A more detailed description of the interface grammar semantics can be found in [12, 11].

4. INTERFACE GRAMMAR COMPILER

In the previous sections we discussed the specification of server or client interfaces of a web service using interface grammars. Now that we have these interface grammars, we would like to generate an actual web service driver and a web service stub in order to perform client and server verification. We achieve this using our interface compiler.

Given an interface specification for a web service server our interface compiler generates a web service stub, and given an interface specification for a web service client (such as the one shown in Figure 2) our interface compiler generates a web service driver.

A web service stub generated by our interface compiler is a top-down parser that parses the sequence of incoming SOAP requests. A service driver generated by our interface compiler, on the other hand, is a language generator which generates a sequence of SOAP requests based on the interface grammar.

In Figure 3 we show the structure of a generic stub/driver generated by our interface compiler. Here the semantics of **choose** and **fail** depend on the environment we are running in. In a conventional JVM, we could have **choose** randomly select one of the options and **fail** throw an exception. When running in a model checker like Java PathFinder[7], **choose** and **fail** hook into the model checker's internal backtracking to exhaustively explore the entire state space.

In order to improve the efficiency of the generated stub/driver, it is worthwhile to pre-compute some information that might be useful in choosing the next production. For example, we may encounter circumstances where we know what the next terminal is—we may have just gotten an incoming method call—and this information could be helpful in picking the next production. Accordingly, our interface compiler transforms the input interface grammar into an LL(1) parse table. This way, in the event that we do know what the next terminal is—that is that we have a symbol of lookahead—we can avoid choosing a branch that is clearly wrong. If we cannot determine that symbol of lookahead, then we fall back to the original definition of **choose**.

$$
\begin{array}{rcll}
\mathit{start} & \rightarrow & \mathit{search}(\text{asin}); \mathit{cart}(\text{asin}) & (1)\\
& | & \epsilon & (2)\\
\mathit{search}(\text{asin}) & \rightarrow & !\textsc{search}(); ¡\textsc{search}(\text{asin}); \mathit{search}'(\text{asin}) & (3)\\
\mathit{search}'(\text{asin}) & \rightarrow & !\textsc{search}(); ¡\textsc{search}(\text{asin}); \mathit{search}'(\text{asin}) & (4)\\
& | & \epsilon & (5)\\
\mathit{cart}(\text{asin}) & \rightarrow & !\textsc{create}(\text{asin}); ¡\textsc{create}(\text{cart}, \text{item}); \mathit{permute}(\text{cart}, \text{item}); \mathit{clear}(\text{cart}) & (6)\\
& | & \epsilon & (7)\\
\mathit{permute}(\text{cart}, \text{item}) & \rightarrow & !\textsc{get}(\text{cart}); ¡\textsc{get}(\text{cart}); \mathit{permute}(\text{cart}, \text{item}) & (8)\\
& | & \langle\!\langle \textsc{choose}\, n > 0 \rangle\!\rangle; !\textsc{modify}(\text{cart}, \text{item}, n); ¡\textsc{modify}(\text{cart}); \mathit{permute}(\text{cart}, \text{item}) & (9)\\
& | & !\textsc{modify}(\text{cart}, \text{item}, 0); ¡\textsc{modify}(\text{cart}) & (10)\\
& | & \mathit{search}(\text{asin}); \mathit{permute}'(\text{cart}, \text{asin}); \mathit{permute}(\text{cart}, \text{item}) & (11)\\
& | & \epsilon & (12)\\
\mathit{permute}'(\text{cart}, \text{asin}) & \rightarrow & [\![\text{asin} \notin \text{ran}(\text{cart})]\!]; !\textsc{add}(\text{cart}, \text{asin}); ¡\textsc{add}(\text{cart}, \text{item}); \mathit{permute}(\text{cart}, \text{item}) & (13)\\
& | & \epsilon & (14)\\
\mathit{clear}(\text{cart}) & \rightarrow & !\textsc{clear}(\text{cart}); ¡\textsc{clear}(\text{cart}); \mathit{clear}(\text{cart}) & (15)\\
& | & !\textsc{get}(\text{cart}); ¡\textsc{get}(\text{cart}); \mathit{clear}(\text{cart}) & (16)\\
& | & \mathit{search}(\text{asin}); \mathit{clear}'(\text{cart}, \text{asin}) & (17)\\
& | & \epsilon & (18)\\
\mathit{clear}'(\text{cart}, \text{asin}) & \rightarrow & [\![\text{asin} \notin \text{ran}(\text{cart})]\!]; !\textsc{add}(\text{cart}, \text{asin}); ¡\textsc{add}(\text{cart}, \text{item}); & \\
& & \mathit{permute}(\text{cart}, \text{item}); \mathit{clear}(\text{cart}) & (19)\\
& | & \epsilon & (20)
\end{array}
$$

Figure 2: Interface grammar for an AWS-ECS client

```
stack ← [start, ⊥]
while stack ≠ [⊥] do
    o || stack ← stack
    if o = nt(v_1, ..., v_n) then
        Choose a production P of o;
        a = ⟨⟨bind o's arguments to the values of v_1, ..., v_n⟩⟩;
        stack ← [a] || P || stack
    else if o = [[p]] ∧ ¬p then
        fail
    else if o = ⟨⟨a⟩⟩ then
        a
    else if o = ?m(v_1, ..., v_n) then
        Receive the incoming method call m;
        Bind m's arguments to v_1, ..., v_n
    else if o = ¿m(v_1, ..., v_n) then
        Return from m with the values of v_1, ..., v_n
    else if o = !m(v_1, ..., v_n) then
        Call m with the values of v_1, ..., v_n
    else if o = ¡m(v_1, ..., v_n) then
        m returns;
        Bind m's return values to v_1, ..., v_n
```

Figure 3: Generic stub/driver pseudocode generated by the interface compiler

Since our interface grammars are not necessarily LL(1) (or even context free) it is not guaranteed that LL(1) parse table will always provide a single choice. Moreover we need to augment the LL(1) parse tables with the appropriate mechanisms to handle the semantic predicates. The details of this process is discussed in our earlier work [11]. Even for cases where the interface grammar is not LL(1), the LL(1) table does help substantially in directing our stubs/drivers.

5. CLIENT VERIFICATION

To demonstrate the value of our approach, we have performed two distinct classes of experimentation. The first, the client verification we detail here, involves verification of a demonstration client for the Amazon E-Commerce Service (AWS-ECS). We generate a stub for the SOAP communication layer so that we can verify the client without connecting to the AWS-ECS and without any network communication. The second, the server verification, involves connecting directly to AWS-ECS itself and checking the AWS-ECS implementation, which we detail in Section 6.

5.1 Amazon Web Service Client

The AWS-ECS client used here in our experiments is a demonstration of programming technique written by Amazon. It is called the AWS-ECS 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 for us in demonstrating the bug finding capabilities of our approach.

The client consists of a Swing GUI that serves as a thinly veiled interface to the AWS-ECS methods. To verify this client, we wish to use JPF, which cannot handle GUIs; accordingly we have written by hand a simple driver that explores several areas that we wish to

Type	Time (in sec)	Memory (in kB)
Typechecking failure	12.5	25,208
Nonsensical data	11.1	25,208
Uncorrelated data	20.8	43,360

Table 1: Input validation errors for the Client.

test. We can classify these areas into two major groupings: 1) input errors that the client ought to be catching but doesn't, and 2) control flow errors that represent execution sequences that are locally valid but globally wrong. An example of the former is passing a string when AWS-ECS expects an integer; an example of the latter is trying to modify contents of the cart after the cart has been cleared. A proper AWS-ECS client should catch these errors and prompt the user to provide appropriate input instead of passing erroneous requests to AWS-ECS. If such input validation is not done at the client side, the user would either see a cryptic error message sent back from the AWS-ECS, or, worse yet, the client may terminate the session (or even crash) based on the error message sent by AWS-ECS. By providing erroneous user input and control sequences to the client, our goal is to discover input and control flow validation errors at the client side.

5.2 Input Errors

To begin, we analyze three input errors that the client, were it doing proper input validation, would catch. In actual execution, one of these (the typecheck failure) would be caught by the Axis communication layer, but the other two would be communicated to AWS-ECS, which would refuse to execute them. The data we gathered for these three errors is summarized in Table 1.

The typechecking failure here exploits a failure of the client to check that the XML Schema type of some of its inputs is actually valid. This comes up when a user enters a string, when an integer is expected. This does not get caught by the Java compiler, as the data in question remains a string until it reaches the serialization layer in the code. In actual execution this would be caught by Axis.

The nonsensical data failure here attempts to add a nonexistent item to a nonexistent cart with a bad checksum. Since this is a syntactically valid request, it would make it all the way to Amazon's servers before being rejected, whereas our stub catches it much earlier.

Finally, the uncorrelated data failure here involves two method calls. The method calls are in the correct sequence and would constitute a valid sequence, except for the fact that the data associated with the two calls is completely uncorrelated. Specifically, we search for an item and then attempt to add another, nonexistent, item to the cart. This again would ordinarily make its way all the way to Amazon.

In Table 1 we show the results of client verification using the JPF model checker and the server server stub that is automatically generated by our interface compiler. Note that executing a verification task like this one without the automatically generated server stub is almost impossible with the existing model checking tools like JPF. Without the server stub, the client has to send a SOAP request to Amazon's servers, which must execute its implementation of the corresponding operation and send the result back. In our framework, the calls to the AWS-ECS are replaced with calls to the server stub. When the server stub receives an incoming call, it executes the semantic predicates and actions that correspond to the operation that is called, and returns the result.

5.3 Control Flow Errors

The preceding errors are useful, but do not demonstrate the full scope of our approach. Accordingly, we have coded the client driver to call AWS-ECS methods in an undirected fashion. The driver first initializes the cart, and then proceeds to call available methods in no particular order. Some call sequences can be perfectly valid executions, but some of them can represent errors. For example, modifying the contents of the cart after clearing it is nonsensical. An appropriate AWS-ECS client would detect such errors by storing some state information at the client side and warning the user against such erroneous requests. In our framework, these types of errors are caught during client verification since these errors trigger semantic predicate violations in the interface grammar.

Type	Depth	Time (sec)	Memory (kB)	Errors
To first error	2	31.8	43,360	0
To first error	3	64.2	62,084	1
To first error	4	49.6	73,456	1
To first error	5	57.3	73,456	1
All errors	2	31.8	43,360	0
All errors	3	77.3	62,084	2
All errors	4	266.8	111,816	15
All errors	5	862.6	229,872	68

Table 2: Control flow validation errors for the Client.

The data we gathered from these runs is summarized in Table 2. Some important details concerning these: we have run each test twice, once until JPF detected the first erroneous path, and then once more discovering all possible erroneous paths. With a depth of 2 our driver is incapable of going wrong; accordingly the first error and all errors data for that depth are the same. At a depth of 3, our analysis takes longer to detect the error than it does for a depth of 4; this is because the first paths our driver executed at depth 3 were in fact correct, and some backtracking had to occur before an error could be detected. By contrast, the timing data for runs detecting all errors follow the exponential time increase one would expect.

6. SERVER VERIFICATION

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. This driver is essentially a sentence generator for the input interface grammar.

The basic sentence generator algorithm is the same algorithm used for all interface grammars, as shown in Figure 3. Because we are generating a driver for server verification, this becomes a top-down sentence generation algorithm that starts with the start symbol and generates a leftmost derivation by applying a production rule to the non-terminal symbol at the top of the stack until the stack is empty. Note that this algorithm generates the sentences on-the-fly, i.e., while generating a sentence, the algorithm is also executing the corresponding test case by making calls to the target web service. The key step here is choosing the next production. We experimented with two approaches for the **choose** function: 1) random sentence generation and 2) Purdom's algorithm.

6.1 Random Sentence Generation

A random sentence generator chooses the next production in the sentence generation algorithm randomly. In order to assess the effectiveness of this random testing approach, we measured the following coverage criteria:

- *Non-terminal coverage:* If we generate sentences randomly, how many sentences do we need to generate in order to cover all the non-terminals and how long does that take? Note that, 100% non-terminal coverage is achieved when all the non-

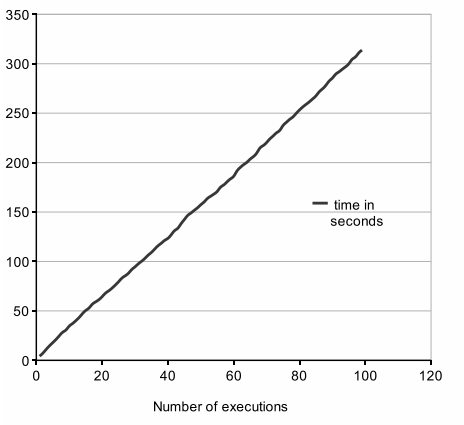

Figure 4: The amount of time it takes to test Amazon's AWS-ECS implementation using the test sequences generated by the random sentence generator

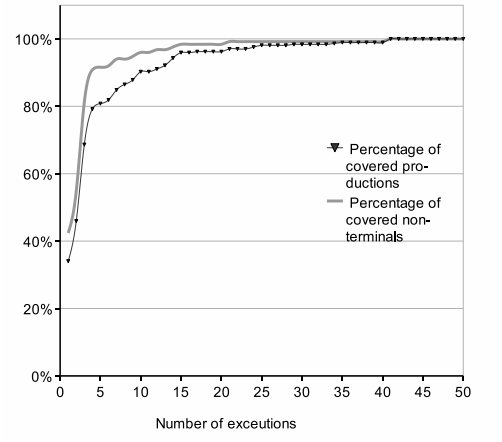

Figure 5: Non-terminal and production coverage obtained using the random sentence generator

terminals appear in derivation of some sentence that has been generated so far.

- *Production coverage:* If we generate sentences randomly, how many sentences do we need to generate in order to cover all the productions and how long does that take? Note that, 100% production coverage is achieved when all the productions are used in some sentence that has been generated so far.

In order to do these measurements we ran ten tests. In each of these tests, we ran the generator until it generated 100 sentences, and then we did the measurements. Finally, we took the averages of these ten measurements. Figures 4 and 5 show the results of our experiments. In Figure 4, we see that the verification time increases linearly with the number of sentences. Figure 5 shows that the full production and nonterminal coverage is achieved after generating 41 sentences on the average. Generating 41 sentences and executing the corresponding 41 test sequences takes about 127 seconds. The generation of the sentences and the execution of the test sequences are done at the same time. 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.

6.2 Purdom's Algorithm

Purdom's algorithm [16] is a sentence generation algorithm that, given a context free grammar, generates a small set of short sentences that guarantees production coverage (i.e., while generating these sentences each production is used at least once). It was developed as a test case generation technique for testing parsers. In this work, we use Purdom's algorithm to generate sentences from a given interface grammar to test the target web service. The generated sentences consist of terminals that correspond to the web service's SOAP requests. Using Purdom's algorithm we can guarantee production coverage (and consequently, terminal and nonterminal coverage).

Given a context free grammar, Purdom's algorithm tries to use all the productions for all the non-terminal symbols. We use a top-down version of Purdom's algorithm [13] where sentences are generated on-the-fly, while executing the test sequences that correspond to the generated sentences.

To apply Purdom's algorithm, we use the same basic top-down sentence generator as in Figure 3, modifying only **choose**. For **choose**, Purdom's algorithm picks a production not only for the non-terminal symbol in the top of the stack, but also all the other non-terminals that might still have productions that are not covered. Purdom's algorithm keeps track of two types of nonterminals: 1) nonterminals that have uncovered productions, and 2) nonterminals that might help in reaching a nonterminal with uncovered productions. If a nonterminal does not fall in either of this categories, then it is marked as *finished*. When a *finished* nonterminal appears in the top of the stack, Purdom's algorithm picks the shortest derivation for that nonterminal. This way Purdom's algorithm tries to keep the lengths of the generated sentences short.

We ran Purdom's algorithm on the AWS-ECS grammar. Purdom's algorithm generated 5 sentences that covered all the productions. These 5 sentences were 2, 11, 58, 33 and 16 derivations long and contained 0, 2, 10, 5 and 2 SOAP requests, respectively. The average derivation length was 24 and the average number of SOAP requests was 3.8. It took 20 seconds to generate these 5 sentences and to execute the test sequences that correspond to these sentences.

6.3 Errors in Amazon Web Service

During the sever verification experiments we discovered two errors. These errors correspond to mismatches between the interface grammar specification and the AWS-ECS implementation.

Error 1: Our initial reading of the AWS-ECS specification led us to believe that it was okay to send multiple ADD requests for the same ASIN. We believed that this would lead to multiple lines in the cart with distinct item IDs, but not otherwise cause trouble. We learned that this was incorrect when we discovered an assertion violation during server verification, and added guards to the corresponding productions to satisfy this restriction. 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. In the AWS-ECS, the items in a cart are stored in a sequence of sequences, which is mapped by the Java layer to a field cartItems on the Cart. This is an instance of the CartItems type, which in its field cartItem contains an array of CartItem objects. We believed that an empty cart, that is a cart with no items, would have a non null Cart.cartItems that contains an array of zero length. However in the AWS-ECS implementation, this is translated as a null Cart.cartItems, so we were forced to change our semantic predicate accordingly. This issue is

not explicitly stated in the API documentation either, although it is present in the WSDL specification.

The experiments we reported in the previous section were conducted after we changed the predicates mentioned above to fix these two errors. We also conducted experiments in the faulty versions where above errors were present. We ran the random sentence generator ten times, stopping each time as soon as we discovered the bug, and took the averages of both time and number of sentences. For the assertion violation (Error 1) the error was discovered after 3.6 runs and took 2.5 seconds. For the null pointer (Error 2) it took on average 5.2 runs and 10.1 seconds to discover the error.

Purdom's algorithm took 12 seconds and generated three sentences before discovering Error 2. However, Purdom's algorithm did not discover Error 1. This is due to the fact that the sentences generated by Purdom's algorithm do not correspond to the scenario where this error is triggered. This, in a way, demonstrates that production coverage is not a very effective criteria since it missed one of the two bugs in the AWS-ECS.

The errors mentioned above can either be considered an error in the AWS-ECS specification or an error in the AWS-ECS implementation. Eventually the goal of both client and server verification is to catch the semantic mismatches between the client's and server's understanding of the web service interface specification. In our approach this interface specification is the interface grammar specification. During client verification we look for mismatches between the interface grammar specification and client implementation and during server verification we look for mismatches between the interface grammar specification and the server implementation. As our results demonstrate, our approach is effective in identifying both types of mismatches.

7. CONCLUSION

We have proposed and implemented a framework for conducting modular verification of web services based on interface grammars. We use interface grammars to specify the interfaces of web services. Using our interface compiler, these interface grammars are automatically converted to web service stubs/drivers to enable modular verification. We 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.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Languages, (POPL 2005)*, 2005.

[3] Amazon web services. http://solutions.amazonwebservices.com/.

[4] 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.

[5] A. Betin-Can and T. Bultan. Verifiable web services with hierarchical interfaces. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2005*, pages 85–94.

[6] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 248–257, 2004.

[7] 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.

[8] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdziński, and F. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, pages 428–441, 2002.

[9] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings 9th Annual Symposium on Foundations of Software Engineering*, pages 109–120, 2001.

[10] 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.

[11] G. Hughes and T. Bultan. Extended interface grammars for automated stub generation. In *Proceedings of the Automated Formal Methods Workshop (AFM 2007)*, 2007.

[12] G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 39–49, 2007.

[13] B. A. Malloy and J. F. Power. A top-down presentation of purdom's sentence-generation algorithm. Technical Report NUIM-CS-TR-2005-04, National University of Ireland at Maynooth, May 2005.

[14] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.

[15] P. M. Maurer. The design and implementation of a grammar-based data generator. *Software Practice and Experience*, 22(3):223–244, March 1992.

[16] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972.

[17] 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.

[18] Simple object access protocol (soap) 1.1. W3C Note 08, http://www.w3.org/TR/SOAP/, May 2000.

[19] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, 2002.

[20] Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[21] Extensible markup language (XML) 1.0 (second edition). W3C, http://www.w3.org/TR/REC-xml, 2000.

[22] XML Schema part 2: Datatypes. W3C Recommendation, http://www.w3.org/TR/xmlschema-2/, May 2001.