Automated Testing of Web Services Based on Algebraic Specifications

Dongmei Liu, Yuxin Liu, Xin Zhang
School of Computer Science and Engineering
Nanjing University of Science and Technology
Nanjing, 210094, P.R. China
dmliukz@njust.edu.cn, liuyuxin980930@sina.com
njust2048@163.com

Hong Zhu and Ian Bayley
Dept of Comp. and Comm. Technologies
Oxford Brookes University
Oxford OX3 1HX, UK
hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

Abstract—The testing of web services must be done in a completely automated manner when it takes place on-the-fly due to third-party services are dynamically composed to. We present an approach that uses algebraic specification to make this possible. Test data is generated from a formal specification and then used to construct and submit service requests. Test results are then extracted and checked against the specification. All these are done automatically, as required. We present ASSAT (Algebraic Specification-Based Service Automated Testing), a prototype that performs these tasks and demonstrate its utility by applying it to Amazon Web Services, a real-life industrial example.

I. INTRODUCTION

A major problem in service-oriented engineering is that it is difficult to trust third-party services. Testing brings confidence that they will work as expected and has therefore been the subject of much research [1], [2]. However, current techniques are not completely automated, and this is a problem because third-party services are discovered dynamically when human intervention is not possible. We solve the problem of how to achieve complete automation by using formal specification, algebraic specification in particular, as a basis of correctness.

The rest of this paper is organised as follows. Section II briefly reviews existing related works. Section III defines preliminary mathematical notions and the specification language SOFIA. Section IV gives the key algorithms of the proposed technique. Section V presents the prototype tool ASSAT. Section VI reports on a case study using ASSAT with Amazon Web Services. Section VII concludes the paper with a discussion of future work.

II. RELATED WORK

It is widely recognised that formal specification can be used as the basis for automated software testing [3], [4], and so much work has been reported in the literature on combining formal methods with software testing [5]. In this section, we first consider the existing approaches to the formal specification of web services. None of these are algebraically-based, however, so we then discuss the advantages of algebraic specification. We finally review work where algebraic specification has been applied to testing software in general rather than limited to web services. The extension to web services is novel so our review on related work is necessarily broader in scope.

A. Formal Specification of Web Services

Existing work on formal specification of web services can be divided into two types. One type uses formal notations indirectly by translating from the original service descriptions in WSDL, OWL-S, BPEL and/or WSMO [1], [6], [7]. This often requires human input, however, and so cannot be automatic as required, because service descriptions are not semantically verifiable so extra semantic information must be added.

The other type employs formal notations directly but only for specifying behaviour. Examples include:

- finite state machines (FSMs) and their variants and extensions, such as extended FSMs [8], Stream X-Machines [9], and protocol state machines (PSMs) [10],
- labelled transition systems and process algebra, such as symbolic labelled transition system STL [11], and
- various kinds of Petri-nets, such as [12].

Behaviour-based formal specifications like these can specify valid sequences of service invocations, for example, but they are weak on functional correctness. Even where test cases can be produced for individual services, it is not known how to turn these into test cases for composed services. A method is proposed for Petri nets in [12], in the context of cloud computing, but as it has not been implemented, it is unclear whether it is feasible.

B. Algebraic specification

Algebraic specification was first proposed in the 1970s as an implementation-independent specification technique for abstract data types [13], [14]. Since then, it has been extended to concurrent systems, state-based systems, software components and service-oriented systems. The theoretical foundations have likewise moved from initial algebras, to final algebras, behavioural algebras [15] and co-algebras [16]–[19].
The qualities of algebraic specifications that make them more suited to web services than other formal specifications are as follows. They are:

- independent of implementation detail, which is appropriate, because no such detail will be available about third-party services,
- highly modular in a manner that suits flexible composition, as is also required by service-oriented engineering,
- easily translatable into ontology-based semantic descriptions, facilitating registration and then dynamic discovery and support [20], [21],
- suitable for specifying dynamic behaviour too, when extended to co-algebraic specifications,
- written in a notation that is easy to learn and to understand, according to empirical studies such as [22]–[24].

Moreover, as we shall see, algebraic specifications allow the whole testing process to be automated, including test case generation, test execution and test oracles to determine the correctness of test results. This has already been demonstrated with objects in OO software and with software components [25].

C. Automated Testing Based on Algebraic Specifications

The most closely related works, applying algebraic specifications, to entities other than web services, are as follows:

- Gannon et al.’s work [26] and Gaudel et al.’s work of testing tools for testing procedural programs [27],
- Frankl and Doong’s work of LOBAS specification language and ASTOOT tool [28], and Hughe et al.’s DAI STISH system [29] for testing OO software, and

The theoretical foundations of these systems and their implementation techniques have been studied in [31]–[34]. We now summarise the key techniques underlying the latter.

In the context of software testing, each ground term of a given signature has two interpretations: it is both a sequence of operation invocations and a value. So to check whether an equation is satisfied, substitute test data for each of the variables and then invoke operations to calculate the left-hand and right-hand sides. If the two are equal, the implementation is correct on the test case; otherwise it is not and there are errors.

Although the basic idea is simple and the first testing tool was developed in the 1980s [26], significant work was required to enable the automated testing of procedural programs [27], OO programs [28], [29] and software components [25], [30]. These techniques cannot be applied immediately to web services, however. They all rely on an ability to create and initialise arbitrary instances of the entity under test, be it an abstract data type, object instantiation of a class, or a software component. In particular, the state of the object before the operations must be copied and stored for comparison with the state after the operations. This is not possible with web services so in this paper we propose a set of techniques to resolve these problems.

III. PRELIMINARIES

In this section, we briefly review the mathematical structure of algebraic specification on which our specification language SOFIA is formally defined [35], [36].

A. Algebraic Structures

We regard a service-oriented system as consisting of a collection of units, each with a unique identifier, called the sort name. There are two ways a unit can be constructed from another: extension and usage. As in [20], [21], we assume that the specification of a software system is well-structured in the following sense.

- Each type of software entity, each type of real-world entity and each type of real-world concept is specified by a corresponding specification unit with a unique name.
- Any extension or usage relationship between software entities, real-world entities and concepts has a corresponding relationship between specification units.

A specification is a triple \( (\mathbf{Sp}, \Sigma, \mathbf{Ax}) \), where

1) \( \mathbf{Sp} = \langle S, \succ, \bowtie \rangle \), where \( S \) is a finite set of sorts, \( \succ \) and \( \bowtie \) are the uses and extends relations on \( S \), respectively;
2) \( \Sigma = \{ \Sigma^s | s \in S \} \) is a set of unit signatures indexed by \( s \), so where each unit signature \( \Sigma^s \) defines a set of typed operators on \( s \);
3) \( \mathbf{Ax} = \{ Ax^s | s \in S \} \) is a finite set of axiom sets indexed by \( s \), so each axiom set \( Ax^s \) defines the semantics of the operators on \{ \( x \in S \mid s \succ x \lor x = s \) \} and the axioms describe the properties that these functions must satisfy.
4) for all \( s \) and \( s' \in S \), \( s \succ s' \) implies that \( \Sigma^s \subseteq \Sigma^s' \) and \( Ax^s \subseteq Ax^{s'} \).

For each \( s \in S \), \( (\Sigma^s, Ax^s) \) is called the specification unit for sort \( s \).

We now define the notion of unit signature to represent the structure of software units as follows. Let \( X \) be a finite set of symbols. We write \( X^* \) to denote the set of finite sequences of symbols in \( X \). In the sequel, we use \( W_s \) to denote \( \{ x \in S | s \succ x \lor x = s \}^* \).

Definition 1: (Unit Signature)

The unit signature \( \Sigma^s \) for a sort \( s \) consists of a finite family of disjoint sets \( \Sigma^s_{w,w'} \) indexed by pairs of units \( (w, w') \) with \( w \) and \( w' \in W_s \). Each element \( \varphi \) in \( \Sigma^s_{w,w'} \) is an operator symbol of type \( w \rightarrow w' \), where \( w \) is the domain type and \( w' \) the co-domain type of the operator.

Such operators can be classified as constants, attributes, and general operations as follows:
1) φ is a constant, if \( w = \emptyset, \) \( w' = (s) \).
2) φ is an attribute, if \( w = (s), w' = (s'), \) and \( s \succ s' \).
3) otherwise, φ is a general operation.

In the sequel, we will write \( \Sigma^c, \Sigma^v, \) and \( \Sigma^g \) for the subsets of \( \Sigma^s \) that contain the constants, the attributes and the general operations, respectively, so that \( \Sigma^s = \Sigma^c \cup \Sigma^v \cup \Sigma^g \).

Let \( \text{Sp}, \Sigma \) be a given system signature and \( s \in S \) be any given sort. We defined the notion of valid terms in \([20], [21]\) that can be used in the specification unit of sort \( s \) as \( s \)-terms. Each \( s \)-term is also typed and its type is \( w \in W_s \).

An equation in a specification unit of sort \( s \) has the form

\[ \tau = \tau', \text{ if } c_1 = d_1, \ldots, c_n = d_n. \]

where \( \tau \) and \( \tau' \) are \( s \)-terms of the same type, \( c_i \) and \( d_i \) are \( s \)-terms of the sort \( s_i \) such that \( s \succ s_i \) for all \( i = 1, 2, \ldots, n \), and \( c_1 = d_1, \ldots, c_n = d_n \) are the conditions. In our theory, we extend the conditional equation by using any comparison operators including \( >, \leq, =, \geq \) in the conditions. So a general conditional equation in specification unit of sort \( s \) has the form

\[ \tau = \tau', \text{ if } c_1 R_1 d_1, \ldots, c_n R_n d_n. \]

where \( R_i \) is a comparison operator.

An axiom set defined in a specification unit of sort \( s \) describes the properties that its operators are required to satisfy. An axiom is a set of conditional or unconditional equations with all variables in these equations universally quantified at the outermost. Formally, we have the following definition.

**Definition 2:** (Axiom Set)

The axiom set \( Ax^s \) for sort \( s \) consists of a finite set of axioms. Each element \( ax^s \in Ax^s \) is an ordered pair \((GV^s_i, E^s_i)\) where

1) \( GV^s_i \) is a finite set, whose elements are the variables declared. These variables are global variables that occur in axioms \( ax^s \).

2) \( E^s_i = \{ (LV^s_{i,j}, e^s_{i,j}) \} \) is the set of conditional equations of axiom \( ax^s \). Each element in \( E^s_i \) is an ordered pair \((LV^s_{i,j}, e^s_{i,j})\), where \( LV^s_{i,j} \) is the set of local variables declared within equation \( e^s_{i,j} \), and each element \( lv^s_{i,j,k} \) is declared as the form \( lv^s_{i,j,k} = \tau^s_{i,j,k} \), \( \tau^s_{i,j,k} \) is a \( s \)-term, and \( e^s_{i,j} \) is a conditional or an unconditional equation. \( LV^s_{i,j} \) is empty if there are no local variables.

**B. Semantics of Algebraic Specification**

We now define the semantics of algebraic specifications by defining what it means for an implementation to be correct with respect to a specification. In general, an implementation of a specification is a mathematical structure that realises the operators in the signature and satisfies the axioms.

**Definition 3:** (Algebra)

Given a system signature \( \text{Sp}, \Sigma \), a \((\text{Sp}, \Sigma)\)-algebra \( A \) is a mathematical structure \((A, \mathcal{F})\) that consists of a collection \( A = \{ A_s | s \in S \} \) of sets indexed by \( s \), and a collection \( \mathcal{F} \) of functions indexed by \( (w, w') \), where \( w, w' \in W_s, s \in S \) such that for each operator \( \varphi : w \rightarrow w' \), the function \( f_{\varphi} \in \mathcal{F} \) has domain \( A_w \) and co-domain \( A_{w'} \), where \( A_w = A_{s_1} \times \cdots \times A_{s_n} \), when \( u = (s_1, s_2, \ldots, s_n) \). □

The evaluation of a \( s \)-term in an algebra depends on the values assigned to the variables that occur in the \( s \)-term. Such an assignment \( \alpha \) of variables \( V_s \), \( s \in S \), in an algebra \( A \) is a function from \( V_s \) to \( A_s \).

**Definition 4:** (Evaluation of \( s \)-terms in an algebra)

Given an assignment \( \alpha \), the evaluation of a \( s \)-term \( \tau \) in an \((\text{Sp}, \Sigma)\)-algebra \( A = (A, \mathcal{F}) \), written \( \text{Ev}_\alpha (\tau) \), is defined as follows.

1. \( \text{Ev}_\alpha (v) = (\alpha (v)) \)
2. \( \text{Ev}_\alpha (\varphi (\tau)) = f_{\varphi} (\text{Ev}_\alpha (\tau)) \)
3. \( \text{Ev}_\alpha (\tau_1, \ldots, \tau_n) = \langle \text{Ev}_\alpha (\tau_1), \ldots, \text{Ev}_\alpha (\tau_n) \rangle \)
4. \( \text{Ev}_\alpha (\tau) = V_k \), where \( \text{Ev}_\alpha (\tau) = \langle V_1, \ldots, V_n \rangle \), \( 1 \leq k \leq n \). □

**Definition 5:** (Satisfaction)

Let \( e \) be an equation. Then an \((\text{Sp}, \Sigma)\)-algebra \( A = (A, \mathcal{F}) \) satisfies \( e \), written \( A \models e \), if for all assignments \( \alpha \), we have that \( \text{Ev}_\alpha (\tau) = \text{Ev}_\alpha (\tau') \) whenever \( \text{Ev}_\alpha (c_i) R_i \text{Ev}_\alpha (d_i) \) is true for all \( i = 1, 2, \ldots, n \).

Let \( \mathcal{E} = (\text{Sp}, \Sigma, \text{Ax}) \) be a specification. An \((\text{Sp}, \Sigma)\)-algebra \( A = (A, \mathcal{F}) \) satisfies specification \( \mathcal{E} \), written \( A \models \mathcal{E} \), if for all equations \( e \) in \( \text{Ax} \), we have that \( A \models e \). □

**C. The SOFIA Specification Language**

SOFIA is a new algebraic specification language based on the algebraic structure described above. Here, we give a brief introduction to the language. The readers are referred to [35] for the reference manual.

The overall structure of a SOFIA specification is a collection of specification units. A unit can be split into two parts: a **Signature** unit, to define the signature, and an **Axiom** unit, to define the axioms that apply to the signature unit. The users can also define auxiliary functions and concepts in a **Definition** unit. More formally, in BNF notation we have:

```
<Specification> ::= <Unit>*
<Unit> ::= <Spec Unit> | <Signature Unit>
| <Axiom Unit> | <Definition Unit>
```

The “extends” and “uses” relations between specification units are declared in clauses introduced with the keywords extends and uses, as shown below.

```
<Spec unit> ::= Spec <Sort Name> [<Observability>];
[<extends <Sort Names>]>[<uses <Sort Names>]
| <Signature> | [<Axioms>]  End
```

SOFIA also declares if a software entity is observable in the sense that its states or values can be directly tested for
equality; otherwise, its states or values have to be checked by other means, e.g. through observers. SOFIA explicitly declares the three kinds of operators mentioned earlier in this section using keywords Const for constants, Attr for attributes, and Operation for general operators. For example, here is a SOFIA specification for Stack.

Spec Stack; uses Int, Real, Bool;
Const: nil;
Attr length: Int; isEmpty: Bool; top: Real;
Operation
Push(Stack,Real): Stack;
Pop(Stack): Stack;
Axiom
For all x: Real, s: Stack that
s.Push(x).Pop = s;
s.Pop.length = s.length-1, if s.length > 0;
s.length = 0, if s.isEmpty = True;
s.isEmpty = False, if s.length > 0;
End

IV. TESTING METHOD

This section describes the testing method.

A. The Testing Process

As shown in Figure 1, the test process consists of three steps:

1) generate test data from the given algebraic specification,
2) construct a sequence of service requests from test data and submit the service requests to the service under test, and
3) receive the service responses from the service under test and check if the responses are correct according to the algebraic specification.

This process is implemented by Algorithm 1.

Algorithm 1 TE: Test Execution

Input: The specification of the web service under test
Output: Testing result of EUT, tr

Step 1: //Initialisation
V\_s\_i,j = \emptyset : T\_s\_i,j = \emptyset
C\_E\_s\_i,j = \emptyset : C\_C\_s\_i,j = \emptyset
Step 2: //Generate test data for GV\_s_i
for each gv\_s_i do
if the sort type of gv\_s_i is primitive then
Generate a value t randomly with satisfaction of constraints
else //Generate test data with a composition structure
t = TDG\_s_i(t)
end if
V\_s\_i,j = V\_s\_i,j \cup gv\_s_i
T\_s\_i,j = T\_s\_i,j \cup t
end for
Step 3: //Check whether test data satisfy the constraints C\_C\_s\_i,j
if there is t \in T\_s\_i,j that doesn’t satisfy the constraints then
goto Step 2
end if
Step 4: //Execute operations of each s-term for declaration of local variable of EUT
if LV\_s\_i,j is not empty then
for each lv\_s_i,j,k \in LV\_s\_i,j do
Substitute test data for variables of s-term \_s\_i,j,k
end if
end if
Step 5: //Execute operations of each s-term of EUT
for each s-term \_s\_i,j,k of EUT do
Substitute t for variable lv\_s_i,j,k of s-
end for
Step 6: //Check whether conditions of EUT satisfy the constraints
for each condition c\_r\_d\_\_ of EUT do
if c\_r\_d\_\_ does not satisfy the constraints then
goto Step 2
end if
end if
Step 7: //Check whether the results are equivalent
if \_s\_i,j = \_s\_i,j then tr = Pass
else tr = Failed
end if
In Algorithm 1, TDG, CC and TRP denote sub-algorithms. TDG generates test data with a composition structure tree, CC constructs constraints of EUT, and TRP performs test execution of the operations of an s-term. The details of these algorithms are given in the following subsections.

B. Test Data Generation

Automated software testing is made possible by the observation that a ground (i.e., variable-free) term corresponds to a sequence of service requests if each operation corresponds to a service request. Therefore, test data can be generated simply by substituting ground terms for variables in axiom equations. The left-hand and right-hand sides will be equivalent if the service satisfies the specification and the constraints are met, if the equation is conditional. These constraints can be either equations or Boolean expressions, as seen respectively in the following examples for the specification of Stack.

s.isEmpty = True, if s.length = 0;
s.isEmpty = False, if s.length > 0;

The constraints must be evaluated first using service requests, as discussed in Subsection IV-D. Here, we focus only on how to generate the ground terms. The method depends on the sort type of the variable being substituted for. If the sort is primitive, random values are used, filtering according to the constraints. For variables of non-primitive sorts, the traditional method is to build up a term by systematically applying constructor operators to constants and random values. We propose here an alternative based on the notions of compositional sort and composition structure trees, which we now define.

A sort is compositional if its specification unit can be constructed by composing other units, including predefined primitive sorts such as String, Integer and Bool. Its state is the aggregation of the states of its components. The composition structure tree is that structure expressed as a tree, and is more formally defined as follows.

Definition 6: (Composition Structure Tree)
The composition structure tree for a compositional sort s is inductively defined as follows.

1) If s is a primitive sort, its composition structure tree is a single node marked with the name of the sort;
2) If s is a sort with no attributes in its signature, its composition structure tree is also a single node marked with the name of the sort;
3) If the signature of sort s contains attributes \( a_1^s, a_2^s, \ldots, a_n^s \), \( n \geq 1 \), and their codomain types are \( s_1, s_2, \ldots, s_n \) respectively, the composition structure tree for sort s has a root node marked with the name of sort s and n sub-trees such that the k-th subtree is the composition structure tree of sort \( s_k \), and the edge from the root node to the k-th subtree is marked with the name of attribute \( a_k^s \).

Note that when the sort contains constants, these are ground terms used as test data. They are especially useful when the constraints are difficult to satisfy. In such a situation, an auxiliary specification unit is constructed, extending the original with appropriate constants. Here is an example where constants ASIN, UPC, SKU, EAN and ISBN are used as various instances of a commercial bar code, where it is infeasible to generate a random value that is meaningful.

Spec CBarCode;
Const id1, id2, id3, id4, id5;
Attr toString: String;
Axiom
  id1.toString = "ASIN";
  id2.toString = "UPC";
  id3.toString = "SKU";
  id4.toString = "EAN";
  id5.toString = "ISBN";
End
End

Algorithm 2 implements the above test data generation method.

Algorithm 2 TDG: Test Data Generation
Input: A sort type s
Output: A composition structured value cv of sort s
\( CE^s = \emptyset \) // Constraints of sort s
for each \( \varphi : s \rightarrow s_k \in \Sigma^s \) do
  if \( s_k \) is primitive then
    Generate a value t randomly with satisfaction of constraints \( CE^s \)
    for each equation e of s do
      Substitute t for \( e \) of all \( s \)-terms of e
      if there is no variable in e then
        \( CE^s = CE^s \cup e \)
      end if
    end for
  end if
else
  TDG(s_k)
end if
if test data cv don’t satisfy the constraints \( CE^s \) then
  TDG(s)
end if

C. Test Result Propagation

After generating test data, we construct service requests according to the s-terms of the equation under test (EUT) and issue the requests via HTTP. For a ground term, the sequence of service requests is essentially the operations \( \varphi \) in the s-terms in left-to-right order. When receiving a response to a service request, we extract the values from response messages and substitute these values together with the test data for variables of the s-terms of the EUT either for the
subsequent request or for checking correctness. Algorithm 3 below gives the details.

Algorithm 3 TRP: Test Result Propagation

Input: A s-term τ with the form x.f₁.f₂....fₙ
Output: A final result of executions of s-term
Push(STs, x) //Stack STs used to save the result of each execution of s-term
for each fi do
  if fi is a general operation then
    Get values from Tᵢᵢᵢ according to inputs of fi
    Construct a HTTP request to execute service operation fi
    Send the HTTP request to the service under test
    Get the response t once it returns to the service requester
    Push(STs, t)
  else if fi is an attribute then
    Get the top value from STs
    Get the value t from STs.top by the keyword fi
    Push(STs, t)
  end if
end for
Return STs.top

D. Evaluation of Test Results

When a sort represents not a primitive but instead structured data, a class, a component, or even a service, the basic idea of generating two ground terms and testing for equality does not work. One solution is to add an observable context to both sides of the equation, making them both observable and comparable. For example, in the axiom of Stack below, the two sides of Equ. (1) are not comparable, but once the attribute length is applied to both sides, giving Equ. (2), the two sides are comparable.

∀x : Real ∀s : Stack s.Push(x).Pop = s; (1)

s.Push(x).Pop.length = s.length; (2)

Observation context is a crucial technique that solves the test oracle problem by re-expressing the comparison of structured data types as comparisons of primitive types, and it is formally defined as follows [33].

Definition 7: (Observable Context)

A context of a sort s is an s-term C with one occurrence of a special variable of sort s. The value of an s-term τ of sort s in the context of C, written as C[τ], is the s-term obtained by substituting τ into the special variable. An observation context oc of sort s is a context of sort s and the sort of the s-term oc is so, where s ≻ so. To be consistent on notations, we write _oc : s → so to denote an observation context oc. An observation context is primitive if so is an observable sort. In such cases, we say that the context is observable. □

The general form of an observable context oc of sort s is:

_0(x₁)....fi(x₁)...._k(x₁)obsk₊₁,...obsk₊m

where f₀, f₁,..., fₖ are the general operations of sort s, s₁,..., sₖ that might be the same and one of the co-domains of fᵢ is sort sᵢ₊₁; obsₖ₊₁,...,obsₖ₊m are the attributes of sort sₖ₊₁,...,sₖ₊m respectively and the co-domain of obsₖ₊j is sort sₖ₊j₊₁. The observable context might not contain any general operation, but must end with an attribute.

Definition 8: ( Observable Context Sequence)

An Observable Context Sequence of a sort s in our specification is the sequential composition _₁₀₁,...,₀c₁₀c₂,...,₀cn, a sequence of observable contexts oc₁, oc₂,..., ocₙ, where _₁₀₁ : s → s₁,...,₀c₁₀c₂ : s₁ → s₂,...,₀cn : sₙ₋₁ → sₙ. An observable context sequence is primitive if sₙ is an observable sort. □

In the case of Stack, for example, the following are observation contexts, all of which are primitive because the predefined sort Int is observable.

_1.top(), _1.Pop.top(), _1.Pop.Pop.top()

_1.length(), _1.Pop.length(), _1.Pop.Pop.length()

Test result evaluation plays two roles in our testing method. First, as discussed above, it checks if a service satisfies the axioms in the specification. Secondly, it checks whether the test data satisfy the constraints on the axioms when the axiom is a conditional equation. If not, the test data are ignored and new test data are generated. To check whether a condition is satisfied, a sequence of service requests that correspond to the condition may also be submitted to the service under test. Algorithm 4 constructs constraints which are used to check if a set of conditions are true. It is used to check if test data satisfy the constraints of an axiom, and also to check if an axiom is satisfied by a service.

Algorithm 4 CC: Correctness Checking

Input: Test data Tᵢᵢᵢ and equations Eᵢᵢᵢ of axiom axᵢᵢᵢ
Output: Constraints CEᵢᵢᵢ and CCᵢᵢᵢ for EUT
for each eᵢᵢᵢ ∈ Eᵢᵢᵢ do
  Substitute test data for variables of all s-terms of eᵢᵢᵢ
  if there is no variable of eᵢᵢᵢ then
    CEᵢᵢᵢ = CEᵢᵢᵢ ∪ eᵢᵢᵢ
  end if
end for
for each condition cᵢᵢᵢRᵢᵢdᵢᵢ of EUT do
  if there is no variable of cᵢᵢᵢRᵢᵢdᵢᵢ then
    CCᵢᵢᵢ = CCᵢᵢᵢ ∪ cᵢᵢᵢRᵢᵢdᵢᵢ
  end if
end for

V. THE PROTOTYPE TOOL ASSAT

This section presents a prototype tool called ASSAT, which stands for Algebraic Specification based Services
Automatic Testing.

ASSAT has been developed in Java to implement our approach described above. As shown in Figure 2, it contains four main components.

1) Specification Parser: parses algebraic specifications written in SOFIA, generates a parse tree, checks the specification is syntactically well-formed, checks the equations in the axioms are type correct.

2) Test Data Generator: as described in the previous section.

3) Test Driver: constructs a sequence of service requests according to the *s-term* of the equations under test, recording the responses as test results.

4) Test Result Evaluator: checks the correctness of the test results and reports errors found to the user.

Figure 2. Overall Structure of the ASSAT Tool

The inputs to ASSAT are the SOFIA specification and the web service under test. Figure 3 shows the interface. On the left is the SOFIA specification and on the right is test data and test results. The Testing Times field is used to input the number of test cases to be used.

VI. CASE STUDY

This section reports a case study of using the testing method and the tool with a real-life industry example: the Amazon Web Services **AWSECommerceService**.

A. Algebraic Specification of **AWSECommerceService**

The Amazon Web services **AWSECommerceService** provides an API for developers to build their own applications. One of its many operations is **ItemSearch**, with many parameters, shown in Table I.

The whole AWSECommerceService API has been specified in SOFIA. The specification has a four-layer structure as shown in Table II. Basic level units such as *item* and *metadata* are constructed using only primitive sorts. Similarly, units at the first level, defining common concepts used by all services such as *errors* and *cart*, are constructed from basic and primitive level units. The second level is on top of the first level and consists of specification units at a higher level of abstraction, such as the requests and responses of various services. Finally, the top-level units specify the service operations of the API. The numbers of specification units at each level is also shown in the table.

For the sake of space, here we define only the single top level specification unit.

**Table I**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Parameter type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWSAccessKeyId</td>
<td>String</td>
<td>Login account of AWS web site</td>
</tr>
<tr>
<td>AssociateTag</td>
<td>String</td>
<td>A tag generated when registering</td>
</tr>
<tr>
<td>Condition</td>
<td>String</td>
<td>Conditions of goods</td>
</tr>
<tr>
<td>Keywords</td>
<td>String</td>
<td>Keyword of goods</td>
</tr>
<tr>
<td>Operation</td>
<td>String</td>
<td>Operation of service</td>
</tr>
<tr>
<td>ResponseGroup</td>
<td>String</td>
<td>Information returned</td>
</tr>
<tr>
<td>SearchIndex</td>
<td>String</td>
<td>Sort of goods</td>
</tr>
<tr>
<td>Service</td>
<td>String</td>
<td>Service transferred</td>
</tr>
<tr>
<td>Version</td>
<td>String</td>
<td>Version number</td>
</tr>
<tr>
<td>MaximumPrice</td>
<td>nonNegativeInteger</td>
<td>The largest price of goods</td>
</tr>
<tr>
<td>MinimumPrice</td>
<td>nonNegativeInteger</td>
<td>The least price of goods</td>
</tr>
<tr>
<td>Timestamp</td>
<td>String</td>
<td>The current timestamp and the format is the ISO-8601 standard format</td>
</tr>
<tr>
<td>Signature</td>
<td>String</td>
<td>A string which is got by encrypting all the parameters before with HMAC</td>
</tr>
</tbody>
</table>

**Table II**

<table>
<thead>
<tr>
<th>Level</th>
<th>Sort</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>AWSECommerceService</td>
<td>1</td>
</tr>
<tr>
<td>Second</td>
<td>BrowseNodeLookupRequest, BrowseNodeLookupResponse, Request</td>
<td>24</td>
</tr>
<tr>
<td>First</td>
<td>OperationRequest, HttpHeaders, Arguments</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>BrowseNodes, BrowseNode, Properties, Children</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Cart, CartItems, SavedForLaterItems, SimilarProducts</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Items, Item, ImageSets, Offers</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>Midcommon, Errors, TopSellers, NewReleases</td>
<td>3</td>
</tr>
<tr>
<td>Basic</td>
<td>Common, Item, MetaData</td>
<td>46</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>127</td>
</tr>
</tbody>
</table>
BrowseNodeLookupRequest):BrowseNodeLookupResponse;  
CartAdd(Common,CartAddRequest,  
CartAddRequest):CartAddResponse;  
CartClear(Common,CartClearRequest,  
CartClearRequest):CartClearResponse;  
CartCreate(Common,CartCreateRequest,  
CartCreateRequest):CartCreateResponse;  
CartGet(Common,CartGetRequest,  
CartGetRequest):CartGetResponse;  
CartModify(Common,CartModifyRequest,  
CartModifyRequest):CartModifyResponse;  
ItemLookup(Common,ItemLookupRequest,  
ItemLookupRequest):ItemLookupResponse;  
SimilarityLookup(Common,SimilarityLookupRequest,  
SimilarityLookupRequest):SimilarityLookupResponse;  
End

Each operator has a set of axioms to characterise its semantics. For reasons of space, we give just those for the ItemSearch operator.

Axiom AWSECommerceService;
For all A:AWSECommerceService, C:Common,  
X: ItemSearchRequest, X1:ItemSearchRequest that
Let res = A.ItemSearch(C,X,X1),  
req = res.items.request  
in X1.Keywords = req.itemSearchRequest.Keywords;  
End
Let res = A.ItemSearch(C,X,X1),  
req = res.items.request,  
code = req.errors.error.Code  
in code = "AWS.MinimumParameterRequirement",  
if X1.Keywords = "";  
End
Let res = A.ItemSearch(C,X,X1),  
req = res.items.request,  
isValid = "False", if X1.Keywords = "";  
End
Let res = A.ItemSearch(C,X,X1),  
req = res.items.request,  
code = req.errors.error.Code  
in code = "AWS.ECommerceService.NoExactMatches",  
if X1.MinimumPrice > X1.MaximumPrice;

where the equations respectively describe the following properties
1) values in responses should be equal to ones in requests by Keywords.
2) there is an error type called AWS.MinimumParameterRequirement if Keywords is null.
3) value of IsValid in responses is False if Keywords is null.
4) there is an error type called AWS.ECommerceService.NoExactMatches in responses if MinimumPrice is greater than MaximumPrice in requests.

B. Testing Results and Analysis

Table III shows the results of automated tested of AWSECommerceService using ASSAT, showing in particular the numbers of test cases in which the web service failed to satisfy the above four axioms; TT denotes the number of test cases generated.

<table>
<thead>
<tr>
<th>Axiom</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>500</th>
<th>800</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equation 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Equation 2</td>
<td>2</td>
<td>14</td>
<td>21</td>
<td>52</td>
<td>128</td>
<td>193</td>
<td>241</td>
</tr>
<tr>
<td>Equation 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Equation 4</td>
<td>3</td>
<td>15</td>
<td>33</td>
<td>39</td>
<td>147</td>
<td>267</td>
<td>329</td>
</tr>
</tbody>
</table>
Most failures occurred with equations 2 and 4 and were caused by the error *Invalid Enumerated Parameter*. Manual checking revealed that parameter *Response Group* was out of the range, and different from the service’s WSDL file. In other words, the implementation of *AWSCommerceService* is not consistent with its WSDL specification and a bug has been found.

Equation 1 and 3 are not affected by the error *Invalid Enumerated Parameter*, and the value of *Keywords* in the responses was as the request asked for. On test cases whose *Keywords* was null, the value of *IsValid* attribute of the responses was *False* as expected and as specified in axiom 3. On one occasion, testing against equation 3 failed but this was caused by a connection timeout.

VII. CONCLUSION AND FUTURE WORK

In this paper, we developed an approach for automated testing of web services based on algebraic specifications written in SOFIA. We presented the details of algorithms for test data generation, test execution and test result evaluation. An automated prototype tool ASSAT has been implemented for testing web services. A case study with a real industrial web service demonstrated the feasibility of the proposed approach.

We are now conducting more experiments to evaluate the fault detection ability and cost-effectiveness of the technique. We also continue to pursue more effective and efficient ways to generate test data using artificial intelligent techniques. To improve the practical usability of our approach, we are also investigating the automatic transformation of ontological descriptions of services into algebraic specifications.

ACKNOWLEDGEMENT

The work reported in this paper is partially supported by National Natural Science Foundation of China (Grant No. 61272420) and Jiangsu Qinglan Project, and partially supported by EU FP7 project MONICA on Mobile Cloud Computing (Grant No.: PIRSES-GA-2011-295222).

REFERENCES


