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, liuyuxin890930@sina.com
njust2048@163.com

Hong Zhu and Ian Bayley
Dept of Comp. and Comm. Technologies
Oxford Brookes University
Oxford OX33 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 DAISTISH system [29] for testing OO software, and
- Zhu et el.'s CASOCC language and CASCAT tool [25], [30] for Java Enterprise Beens software components.

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 $(\mathbf{Sp}, \mathbf{\Sigma}, \mathbf{Ax})$, where

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

For each $s \in S$, (Σ^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 $\{x \in S | s \succ x \vee x = s\}^*$.

Definition 1: (Unit Signature)

The unit signature Σ^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 φ in set $\Sigma^s_{w,w'}$ is an *operator symbol* of type $w \to w'$, 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 Σ^s_C, Σ^s_V and Σ^s_G for the subsets of Σ^s that contain the constants, the attributes and the general operations, respectively, so $\Sigma^s = \Sigma^s_C \cup \Sigma^s_A \cup \Sigma^s_G$.

Let $(\mathbf{Sp}, \mathbf{\Sigma})$ 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-terms is also typed and its type is $w \in W_s$.

An equation in a specification unit of sort s has the form $\tau = \tau'$, where τ and τ' are s-terms of the same type. A typical conditional equation in a specification unit of sort s has the form

$$\tau = \tau', \text{ if } c_1 = d_1, \cdots, c_n = d_n.$$

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

$$\tau = \tau', \text{ if } c_1 R_1 d_1, \cdots, c_n R_n d_n.$$

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 $ax^s_i \in Ax^s$ 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_i.
2) $E^s_i = \{(LV^s_{i,j}, e^s_{i,j})\}$ is the set of conditional equations of axiom ax^s_i. 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 $(\mathbf{Sp}, \mathbf{\Sigma})$, a $(\mathbf{Sp}, \mathbf{\Sigma})$-*algebra* $\mathcal{A}$ is a mathematical structure $(\mathbf{A}, \mathbf{F})$ that consists of a collection $\mathbf{A} = \{A_s | s \in S\}$ of sets indexed by s, and a collection $\mathbf{F}$ of functions indexed by (w, w'), where $w, w' \in W_s$, $s \in S$ such that for each operator $\varphi : w \to w'$, the function $f_\varphi \in \mathbf{F}$ has domain A_w and co-domain $A_{w'}$, where $A_u = A_{s_1} \times \cdots \times A_{s_n}$, when $u = (s_1, s_2, \cdots, s_n)$. □

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 α of variables V_s, $s \in S$, in an algebra $\mathcal{A}$ is a function from V_s to A_s.

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

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

1) $Eva_\alpha(v) = \alpha(v)$;
2) $Eva_\alpha(\varphi(\tau)) = f_{A,\varphi}(Eva_\alpha(\tau))$.
3) $Eva_\alpha(\langle \tau_1, \cdots, \tau_n \rangle) = \langle Eva_\alpha(\tau_1), \cdots, Eva_\alpha(\tau_n) \rangle$;
4) $Eva_\alpha(\tau \# k) = V_K$, where $Eva_\alpha(\tau) = \langle V_1, \cdots, V_n \rangle$, $1 \le k \le n$. □

Definition 5: (Satisfaction)

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

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

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 partial units: 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 = True, if s.length = 0;
  s.isEmpty = False, if s.length > 0;
 End
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:

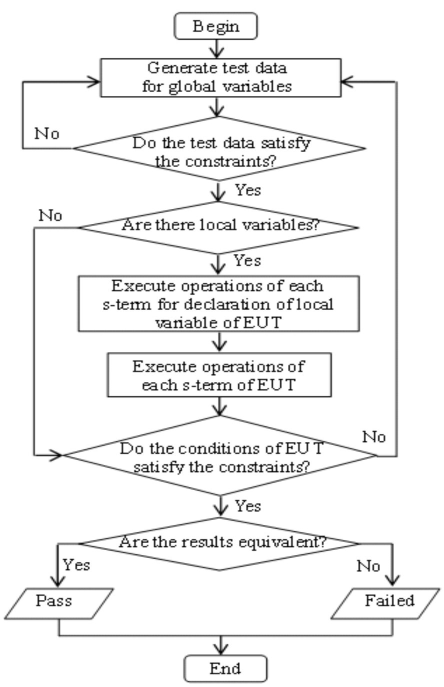

Figure 1. The Testing Process

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
$V^s_{i,j} = \emptyset : T^s_{i,j} = \emptyset$
$CE^s_{i,j} = \emptyset : CC^s_{i,j} = \emptyset$
Step 2: //Generate test data for GV^s_i
for each $gv^s_i \in GV^s_i$ **do**
  if the sort type of gv^s_i is primitve **then**
    Generate a value t randomly with satisfaction of constraints
  else //Generate test data with a composition structure tree
    $t = TDG(the\ sort\ type\ of\ gv^s_i)$
  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
CC($T^s_{i,j}$, E^s_i) //Constrcut constraints for EUT
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 varaible 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* $\tau^s_{i,j,k}$
    $t = TP(\tau^s_{i,j,k})$
    $V^s_{i,j} = V^s_{i,j} \cup lv^s_{i,j,k} : T^s_{i,j} = T^s_{i,j} \cup t$
    Substitute t for variable $lv^s_{i,j,k}$ of *s-terms* of EUT
  end for
end if
Step 5: //Execute operations of each *s-term* of EUT
for each *s-term* τ of EUT **do**
  $t = TP(\tau)$: Substitute t for τ of EUT
end for
Step 6: //Check whether conditions of EUT satisfy the constraints
for each condition $c_i R_i d_i$ of EUT **do**
  if $c_i R_i d_i$ does not satisfy the constraints **then**
    goto Step 2
  end if
end for
Step 7: //Check whether the results are equivalent
if $\tau = \tau'$ **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 (ie 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 $a_1^s, a_2^s, \cdots, a_n^s$, $n \geq 1$, and their codomain types are $s'_1, s'_2, \cdots, s'_n$ 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 $\varphi : s \rightarrow s_k \in \Sigma_A^s$ **do**
    if s_k is primitive **then**
      Generate a value t randomly with satisfaction of constraints CE^{s_k}
      for each equation e of s **do**
        Substitute t for $.\varphi$ of all *s-terms* of e
        if there is no variable in e **then**
          $CE^s = CE^s \cup e$
        end if
      end for
      Assign t to node $s.\varphi$ of cv
    else
      TDG(s_k)
    end if
  end for
  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 φ in the *s-terms* in left-to-right order. When receiving a response to a service request, we extract the values from response message 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 $x.f_1.f_2.....f_n$
Output: A final result of executions of *s-term* τ
  $Push(ST^s, x)$ //Stack ST^s used to save the result of each execution of *s-term*
  for each f_i **do**
    if f_i is a general operation **then**
      Get values from $T^s_{i,j}$ according to inputs of f_i
      Construct a HTTP request to execute service operation f_i
      Send the HTTP request to the service under test
      Get the response t once it returns to the service requester
      $Push(ST^s, t)$
    else if f_i is an attribute **then**
      Get the top value from ST^s
      Get the value t from $ST^s.top$ by the keyword f_i
      $Push(ST^s, t)$
    end if
  end for
  Return $ST^s.top$

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.

$$\forall x : Real\ \forall s : Stack\ s.Push(x).Pop = s; \tag{1}$$
$$s.Push(x).Pop.length = s.length; \tag{2}$$

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 $C[\tau]$, 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 s', where $s \succ s'$. To be consistent on notations, we write $\_.oc : s \to s'$ to denote an observation context oc. An observation context is primitive if s' 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: $\_.f_0(\ldots).f_1(\ldots).\cdots.f_k(\ldots).obs_{k+1}.\ldots.obs_{k+m}$

where $f_0, f_1, \ldots, f_k$ are the general operations of sort $s, s_1, \ldots, s_k$ that might be the same and one of the co-domains of f_i is sort s_{i+1}; $obs_{k+1}, \ldots, obs_{k+m}$ are the attributes of sort $s_{k+1}, \ldots, s_{k+m}$ respectively and the co-domain of obs_{k+j} is sort s_{k+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 $\_.oc_1.oc_2.\ldots.oc_n$, a sequence of observable contexts $oc_1, oc_2, \ldots, oc_n$, where $\_.oc_1 : s \to s_1$, $\_.oc_2 : s_1 \to s_2$, ..., $\_.oc_n : s_{n-1} \to s_n$. An observable context sequence is *primitive* if s_n 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.

```
_.top(), _.Pop.top(), _.Pop.Pop.top()
_.length(), _.Pop.length(), _.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 $T^s_{i,j}$ and equations E^s_i of axiom ax^s_i
Output: Constraints $CE^s_{i,j}$ and $CC^s_{i,j}$ for EUT
  for each $e^s_{i,j} \in E^s_i$ **do**
    Substitute test data for variables of all *s-terms* of $e^s_{i,j}$
    if there is no variable of $e^s_{i,j}$ **then**
      $CE^s_{i,j} = CE^s_{i,j} \cup e^s_{i,j}$
    end if
  end for
  for each condition $c_i R_i d_i$ of EUT **do**
    if there is no variable of $c_i R_i d_i$ **then**
      $CC^s_{i,j} = CC^s_{i,j} \cup c_i R_i d_i$
    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.

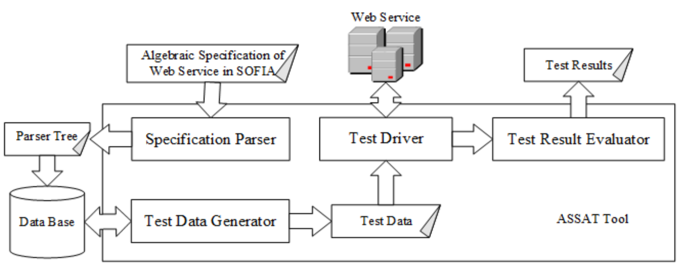

Figure 2. Overall Structure of the ASSAT Tool

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.

```
Spec AWSECommerceService;
uses Common, ItemSearchRequest,ItemSearchResponse,
  BrowseNodeLookupRequest,BrowseNodeLookupResponse,
  CartAddRequest,CartAddResponse,
  CartClearRequest,CartClearResponse,
  CartCreateRequest,CartCreateResponse,
  CartGetRequest,CartGetResponse,
  CartModifyRequest,CartModifyResponse,
  ItemLookupRequest,ItemLookupResponse,
  SimilarityLookupRequest,SimilarityLookupResponse;
Operation
 ItemSearch(Common,ItemSearchRequest,
   ItemSearchRequest):ItemSearchResponse;
 BrowseNodeLookup(Common,BrowseNodeLookupRequest,
```

Table I
PARAMETERS OF ITEMSEARCH

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

Table II
NUMBER OF UNITS IN AWSECOMMERCESERVICE SPECIFICATION

<table>
<tr><td>Level</td><td colspan="2">Sort</td><td colspan="2">number</td></tr>
<tr><td>Top</td><td colspan="2">AWSECommerceService</td><td colspan="2">1</td></tr>
<tr><td>Second</td><td colspan="2">BrowseNodeLookupRequest, BrowseNodeLookupResponse, Request</td><td colspan="2">24</td></tr>
<tr><td rowspan="5">First</td><td>OperationRequest</td><td>HttpHeaders, Arguments</td><td>3</td><td rowspan="5">56</td></tr>
<tr><td>BrowseNodes</td><td>BrowseNode, Properties, Children</td><td>6</td></tr>
<tr><td>Cart</td><td>CartItems, SavedForLaterItems, SimilarProducts</td><td>8</td></tr>
<tr><td>Items</td><td>Item, ImageSets, Offers</td><td>36</td></tr>
<tr><td>Midcommon</td><td>Errors, TopSellers, NewReleases</td><td>3</td></tr>
<tr><td>Basic</td><td colspan="2">Common, Item, MetaDada</td><td colspan="2">46</td></tr>
<tr><td>Total</td><td colspan="2"></td><td colspan="2">127</td></tr>
</table>

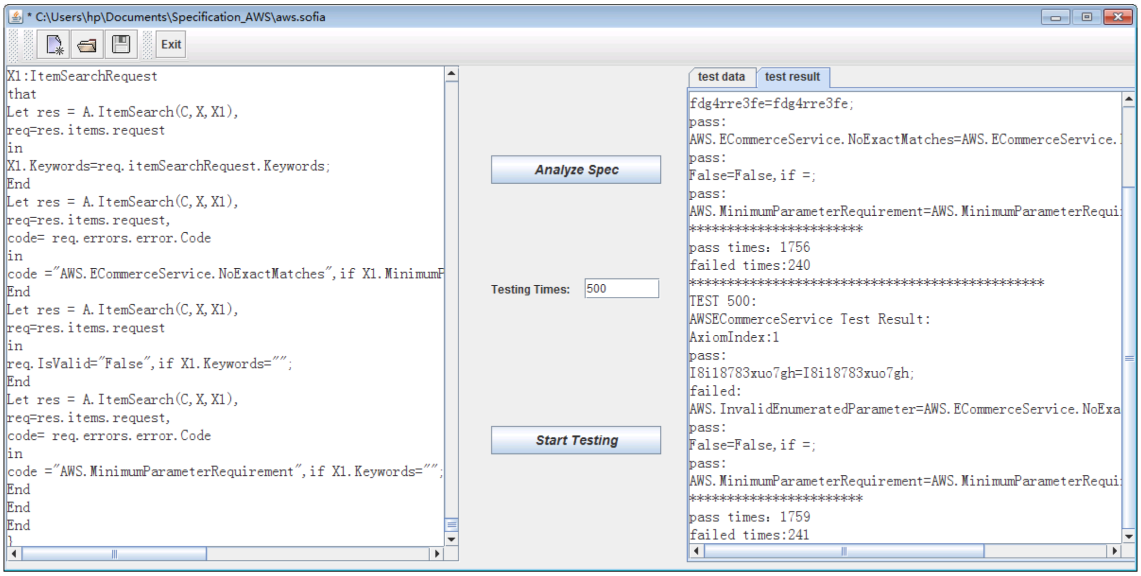

Figure 3. The Interface of ASSAT

```
    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
    in req.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;
    End
  End
End
```

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 *IsVaild* in reponses 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 III
TESTING RESULTS

NOF \ TT / Axiom	10	50	100	200	500	800	1000
Equation 1	0	0	0	0	0	0	0
Equation 2	2	14	21	52	128	193	241
Equation 3	0	0	0	0	0	1	0
Equation 4	3	15	33	59	147	267	329

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 *AWSECommerceService* 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

[1] Bozkurt M. , Harman M. and Hassoun Y. , "Testing and verification in service-oriented architecture: a survey," *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 261–313, 2013.

[2] Canfora G., Di Penta M., "Service-oriented architectures testing: A survey," *LNCS*, vol.5413, pp. 78–105, 2009.

[3] Gaudel, M.-C., "Testing Can Be Formal, Too", in *Proc. of TAPSOFT '95*, Springer-Verlag, pp. 82–96, 1995.

[4] Gaudel, M.-C., "Software Testing Based on Formal Specification", P. Borba et al. (Eds.), in *Proc. of PSSE 2007*, LNCS 6153, pp. 215–242, 2010.

[5] Hierons R M, Bogdanov K, Bowen J P, et al., "Using formal specifications to support testing," *ACM Computing Surveys*, vol. 41, no. 2, pp. 1–76, 2009.

[6] Tahir, A., Tosi, D., Morasca, S.,"A systematic review on the functional testing of semantic web services", *The Journal of Systems and Software*, Vol. 86, pp2877–2889, 2013.

[7] Endo, A. T., and Silva Simao, A., "Formal Testing Approaches for Service-Oriented Architectures and Web Services: a Systematic Review", Technical Report, No. 348, Instituto de Ciencias Matem?aticas e de Computa?c?ao, Brazil, ISSN - 0103-2569, March 2010.

[8] Keum C. S., Kang, S., Ko, I. Y., et al., "Generating test cases for web services using extended finite state machine", *Testing of Communicating Systems*, Springer, pp. 103–117, 2006.

[9] Ramollari, E., Kourtesis, D., Dranidis, D., et al., "Leveraging semantic web service descriptions for validation by automated functional testing", *The Semantic Web: Research and Applications*. Springer, pp. 593–607, 2009.

[10] Bertolino, A., Frantzen, L., Polini, A., et al., "Audition of web services for testing conformance to open specified protocols", *Architecting Systems with Trustworthy Components*. Springer, pp. 1–25, 2006.

[11] Frantzen, L., de las Nieves Huerta M., Kiss, Z. G., et al. "On-the-fly model-based testing of web services with Jambition", *Web Services and Formal Methods*. Springer, pp. 143–157, 2009.

[12] Zhang, X., Wu, C. and Xue, S. "Petri nets based test case selection model for service composition in cloud", in *Proc. of the 4th IEEE Int'l Conf. on Digital Manufacturing and Automation*, pp. 914–917, 2013.

[13] Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B., "Initial algebra semantics and continuous algebras," *Journal of ACM*, vol. 24, no. 1, pp. 68–95, 1977.

[14] Ehrich, H.-D., "On the theory of specification, implementation, and parametrization of abstract data types," *Journal of ACM*, vol. 29, no. 1, pp. 206–227, 1982.

[15] Goguen, J.A. and Malcolm,G., "A hidden agenda," *Theor. Comput. Sci.*, vol. 245, no. 1, pp. 55–101, 2000.

[16] Cîrstea C., "Coalgebra semantics for hidden algebra: Parameterised objects and inheritance," *Recent Trends in Algebraic Development Techniques, 12th International Workshop (WADT'97)*, pp.174–189, 1997.

[17] Rutten J. M. , "Universal coalgebra: a theory of systems," *Theor. Comput. Sci.*, vol. 249, no. 1, pp. 3–80, 2000.

[18] Cîrstea C., "A coalgebraic equational approach to specifying observational structures," *Theoretical Computer Science*, vol.280, no. 1-2, pp. 35–68, 2002.

[19] Bonchi F. and Montanari U., "A coalgebraic theory of reactive systems," *Electr. Notes Theor. Comput. Sci.*, vol. 209, pp. 201–215, 2008.

[20] Liu, D., Zhu, H., and Bayley, I., " From Algebraic Formal Specification to Ontological Description of Service Semantics," in *Proc. of ICWS 2013*, pp. 579–586, 2013.

[21] Liu, D., Zhu, H., and Bayley, I., "Transformation of Algebraic Specifications into Ontological Semantic Descriptions of Web Sservices," *International Journal of Services Computing*, vol. 2, no. 1, pp.58–71, 2014.

[22] Zhu, H. and Yu, B., "An Experiment with Algebraic Specifications of Software Components," in *Proc. of QSIC 2010*, pp. 190–199, 2010.

[23] Liu, D., Zhu, H., and Bayley, I., "Applying algebraic specification to cloud computing–a case study of infrastructure-as-a-service GoGrid," in *Proc. of ICSEA 2012*, pp. 407–414, 2012.

[24] Liu, D., Zhu, H., and Bayley, I., "A case study on algebraic specification of cloud computing," in *Proc. of PDP 2013*, pp. 269–273, 2013.

[25] Yu, B., Kong, L., Zhang, Y., and Zhu, H., "Testing java components based on algebraic specifications," in *Proc. of ICST 2008*, pp. 190–199, 2008.

[26] Gannon J, McMullin P, Hamlet R., “Data abstraction, implementation, specification, and testing,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 3, no. 3, pp. 211–223, 1981.

[27] Bernot G. , Gaudel M.-C. , and Marre B., “Software testing based on formal specifications: a theory and a tool,” *Software Engineering Journal*, vol. 6, no. 6, pp. 387–405, 1991.

[28] Doong R. K. and Frankl, P. G., “The ASTOOT approach to testing object-oriented programs,” *ACM TSEM*, vol. 3, no. 2, pp. 101–130, 1994.

[29] Hughes M., Stotts, D., “Daistish: systematic algebraic testing for OO programs,” in *Proc. ISSTA 1996*, ACM Press, pp. 53–61, 1996.

[30] Kong, L., Zhu, H., and Zhou, B., “Automated testing EJB components based on algebraic specifications,” in *Proc. COMPSAC’07*, vol. 2, pp. 717–722, 2007.

[31] Chen, H. Y. , Tse T. H. , Chan, F. T., and Chen, T. Y. , “In black and white: An integrated approach to class-level testing of object-oriented programs,” *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 250–295, 1998.

[32] Chen, H. Y. , Tse T. H. , and Chen, T. Y., “TACCLE: a methodology for object-oriented software testing at the class and cluster levels,” *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 1, pp. 56–109, 2001.

[33] Zhu, H., “A Note on Test Oracles and Semantics of Algebraic Specifications”, in *Proc. of QSIC 2003*, pp. 91–99, 2003.

[34] Chen, H.Y., and Tse, T.H. , “Equality to Equals and Unequals: A Revisit of the Equivalence and Nonequivalence Criteria in Class-Level Testing of Object-Oriented Software,” IEEE Transactions on Software Engineering, vol. 39, no. 11, pp. 1549-1563, 2013.

[35] Zhu, H., Liu, D., and Bayley, I., “Reference manual of the SOFIA algebraic specification language” Technical Report TR-CCT-AFM-01-2013, Oxford Brookes University, Oxford, UK, 2013.

[36] Liu, D., Zhu, H., and Bayley, I., “SOFIA: An Algebraic Specification Language for Developing Services,” in *Proc. of SOSE 2014*, pp. 70–75, 2014.