Monic Testing of Web Services Based on Algebraic Specifications

Dongmei Liu, Xian Wu, Xin Zhang
School of Computer Science and Engineering
Nanjing University of Science and Technology
Nanjing, 210094, P.R. China
dmliukz@njust.edu.cn, 755368601@qq.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—Web services are designed to be discovered and composed dynamically, which implies that testing must also be done dynamically. This involves both the generation of test cases and the checking of test results. This paper presents algorithms for both of these using the technique of algebraic specification. It focuses in particular on the problem that web services, when they are third-party, have poor controllability and observability, and introduces a solution known as monic floating checkable test cases. A prototype tool has implemented the proposed testing technique and it is applied to a case study with a real industry application GoGrid, demonstrating that the technique is both applicable and feasible.

Keywords—Web Services; Algebraic Specifications; Test Automation; Test Case Generation; Test Oracle.

I. INTRODUCTION

A major challenge of service-oriented software development is to ensure that third-party services dynamically discovered and composed are semantically correct. This means that their functions and behaviours are as expected. Testing is one solution but it must be done on-the-fly and therefore automatically [1]. Existing methods have not yet achieved this automation, so this paper proposes a novel technique called monic algebraic testing.

Section II briefly reviews existing work and identifies the open problems. Section III introduces the basic concepts of algebraic specification and algebraic testing, and introduces a running example, to explain further concepts and illustrate theorems. Section IV introduces our proposed solution. Section V gives the algorithms of test case generation. Section VI reports a case study with GoGrid, a real industrial RESTful web service. Section VII concludes the paper with a discussion of future work.

II. RELATED WORK AND THE OPEN PROBLEM

This section reviews related work in the areas of testing web services, an active area of research for the past decade [2]–[4], and testing software in general based on algebraic specifications.

A. Testing Web Services

Most of the existing work on testing web services are based on definitions in WSDL. Since such definitions are purely syntactic and limited to input and output data types, any testing that uses them as specification cannot ensure semantic correctness. Some works, on the other hand, are based on semantic web services (SWS) specifications. These provide both additional behavioural information and ontological description of the meanings of data and operations using a domain vocabulary. Such information is useful for checking the correct sequencing of messages passed between service requesters and providers, but semantic descriptions based on ontology are not verifiable for functional correctness [5].

It is widely recognised that formal specifications can be used for automated software testing [6], enabling the correctness of a software system and its components to be verified. A large amount of work on integrating formal methods with software testing has been reported in the literature [7]. However, relatively little of this work concerns services, where the standard approach is to derive formal specifications indirectly by translating from service descriptions in WSDL, OWL-S and/or BPEL [2]–[4]. Such translations often require additions to the semantics. This requires human input so the testing cannot be done on-the-fly. Where formal specification is applied directly to web services it is all behaviour-based. Examples include:

- finite state machines and their variants and extensions, such as EFSMs [8], Stream X-Machines [9], and protocol state machines (PSMs) [10].
- labelled transition systems and process algebra, such as symbolic labelled transition systems (STLs) [11].
- various kinds of Petri nets, such as [12]–[14].

The behaviour-based approach, in the context of service composition, defines valid sequences of service invocations, but functional correctness cannot be specified this way so we use algebraic specification instead as it is property-based.
B. Software Testing based on Algebraic Specifications

The algebraic method of specification was first proposed in the 1970s for abstract data types [15], [16]. Since then it has been applied to concurrent systems, state-based systems, software components and service-oriented systems [17], [18]. The theoretical mathematical foundations have expanded from initial and final algebras to behavioural algebras [19] and co-algebras [20], [21]. Both functional and behavioural properties can thereby be specified and it seems that this approach may be suitable for web services [5], [22].

Moreover, algebraic specifications make it possible to automate the whole testing process, including both generating the test cases and checking the test results. Since its introduction in the early 1980s [23], algebraic testing technique has been advanced significantly. The most well-known works include:

- Gaudel et al.’s use of Prolog to implement a tool for testing procedural programs [24],
- Frankl and Doong’s work of LOBAS specification language and ASTOOT tool [25], and Hughe’s DAISTISH system [26] for testing OO software, and
- Zhu et al.’s CASOC language and CASCAT tool [27], [28] for Enterprise JavaBeans software components.
- Chen et al’s studies of the theoretical foundations in the context of object-oriented software [29]–[31]
- Zhu’s study connecting algebraic testing to formal semantics of specification [32].

However, how to apply this to web services for automated testing is still an open problem.

C. The Problem

Consider the following test case for an online banking service.

\[
\text{deposit(acnt, $200);} \quad \text{withdraw(acnt, $100);} \quad \text{balance(acnt)}
\]

\[
= \text{deposit(acnt, $100);} \quad \text{balance(acnt)}
\]

Existing testing techniques would generate the following execution sequence.

1. deposit(acnt, $200);
2. withdraw(acnt, $100);
3. integer x:= balance(acnt);
4. deposit(acnt, $100);
5. integer y:= balance(acnt);
6. IF (x=y) THEN "Pass" ELSE "Error" ENDIF

However, this sequence is wrong as an implementation of the test case, because acnt, denoting the state of the web service, in line 4 is different from acnt in line 1. Existing solutions to this problem are to have two copies of the service, one for each side of the equation, or to reset the state after executing the left side. Unfortunately, neither approach can be applied to web services, because their third-party nature means that such copying and resetting operations are not usually available. Our solution is a new algebraic testing technique called monic testing.

III. PRELIMINARIES

A. Algebraic Specification and SOFIA Language

1) General Concept of Algebraic Specifications: An algebraic specification consists of a signature  \( \Sigma \) and a set of axioms. The signature comprises a set of sorts and a set of operators. Each sort represents the type of a software system entity, such as a data item, object, software component, or even, in our present context of web services, a message passed between services.

Each operator  \( \varphi \in \Phi \) is associated with two lists of sorts: a domain  \( \vec{u} = s_1, \cdots, s_n \), where  \( n \geq 0 \), and a codomain  \( \vec{v} = s'_1, \cdots, s'_m \), where  \( m \geq 1 \). We summarise this as  \( \varphi : \vec{u} \rightarrow \vec{v} \).

The axioms collectively define the semantics of the operators, and each one can be either an equation or a conditional equation. An equation is written in the form  \( \tau = \tau' \), where  \( \tau \) and  \( \tau' \) are terms formed by applying the operators to constants and variables of various sorts. It means that the equation holds for every possible consistent substitution of values for the variables. A conditional equation, on the other hand, is written in the form  \( \tau = \tau' \) if  \( C \), where  \( C \) is a set of equations. It means the equation  \( \tau = \tau' \) holds as above but only if the values satisfy the condition  \( C \).

2) The SOFIA Language: SOFIA, defined in [17], [18], is an algebraic specification language for service-oriented systems. Each specification is modular and therefore made up of one or more units each defining a sort.

A specification in SOFIA is a triple \( (S, \Sigma, Ax) \), where

1) \( S = \langle S, \succ, \succ \rangle \), where  \( S \) is a finite set of sorts, \( \succ \) and \( \succ \) are binary relations on  \( S \) representing the uses and extends relations on  \( S \), respectively, which represent the two ways in which one unit can be constructed from another;

2) \( \Sigma = \{ \Sigma_s | s \in S \} \) is a set of signatures indexed by  \( s \), where  \( \Sigma_s \) is called the unit signature of a sort  \( s \) and has typed operators whose domain and codomain are in \( \{ x \in S \mid s \succ x \vee x = s \} \);

3) \( Ax = \{ Ax_s | s \in S \} \) is a finite set of axiom sets indexed by  \( s \), where the semantics of the operators in  \( \Sigma_s \) are defined by the set  \( Ax_s \cup \bigcup_{s \rightarrow x} Ax_x \).

For each  \( s \in S \), \( (\Sigma_s, Ax_s) \) is called the specification unit for sort  \( s \) and, using BNF notation, has the following syntax.

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

Here, keywords extends and uses indicate that the sorts being defined are connected by the corresponding relations of the same name. SOFIA distinguishes two specialised types of operators: constants and attributes, denoted by keywords Const and Attr. Any operator that is neither is denoted by the keyword Operation. More formally, let  \( \varphi : s_1, \cdots, s_n \rightarrow s'_1, \cdots, s'_m \) be an operator defined in  \( \Sigma_s \).
It is a constant if $n = 0$, $m = 1$, and $s' = s$, but an attribute if $n > 0$, $s \in \{s_1, \ldots, s_n\}$, and $s \notin \{s'_1, \ldots, s'_m\}$.

Here is a specification unit in SOFIA for a single Integer stack, running on a server on a network, and accessed by sending a message for an operation over the network. That operation changes the internal state and sends a message back to the service requestor.

**Example 1:**

1. Spec StackService;
2. uses Integer, Bool, Message;
3. Const: nil;
4. Attr
5. length: Integer;
6. isEmpty: Bool;
7. top: Integer;
8. Operation
9. push(Integer): Message;
10. pop(): Message;
11. Axiom
12. nil.isEmpty = True;
13. nil.length = 0;
14. For all s: StackService that
15. s.isEmpty = True, if s.length=0;
16. s.isEmpty = False, if s.length>0;
17. s.pop().length=s.length-1, if s.length>0;
18. End
19. For all x: Integer, s: StackService that
20. s.push(x).isEmpty = False;
21. s.push(x).top = x;
22. s.push(x).pop = s;
23. s.push(x).length = s.length+1;
24. End
25. End

A test case $(\tau, \tau')$ of ground terms is positive if $\tau$ and $\tau'$ are equivalent and negative otherwise [25]. The latter are redundant, however, because Chen et al. proved that passing all the positive test cases is sufficient for guaranteeing the correctness [31].

**B. Algebraic Testing**

Algebraic testing is based on the observation that each ground term of a signature can be interpreted either as a value or as a sequence of operation invocations. So to check whether an equation is satisfied, we can simply 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 software under test is correct on the test case; if not, there are errors. Researchers have examined how to select test cases, how to check whether an equation holds and how to translate a term into a sequence of operations on the entity under test. We now summarise the existing techniques.

1) **Types of Test Cases: Positive vs Negative:** A test case $(\tau, \tau')$ of ground terms is positive if $\tau$ and $\tau'$ are equivalent and negative otherwise [25]. The latter are redundant, however, because Chen et al. proved that passing all the positive test cases is sufficient for guaranteeing the correctness [31].

2) **Generation of Test Cases: Normalisation vs Instantiation:** There are two approaches for generating positive test cases. One way is to generate a ground term and then to use the algebraic specification to rewrite the term into a normal form, containing constructors and constants. The original term $\tau$ and its normal form $\tau'$ then form a positive test case [25]. This approach assumes that the specification is complete, and requires a tool to support term rewriting. Another simpler way, with neither of these limitations, is to substitute every variable in each axiom with a ground term. Chen et al. [29], [30] proved that both ways are equally effective.

3) **Checking Equality: Observation Context vs Checkable Test Cases:** There are three ways to check a test case $(\tau = \tau')$ when the sort of $\tau$ and $\tau'$ is not observable. One way is simply to implement $=\$, making the sort observable. Another is to test for equality by applying what we call observation contexts to both sides. An observation context of sort $s$ is a term $\theta(x_s)$ of observable sort with one occurrence of a free variable $x_s$ of sort $s$.

Both of these ways require that an original entity be copied to give one copy for each side of the equation. This cannot be done in a component-based system, however, nor can it for third-party web services. Addressing this problem, the checkable test case technique of Kong et al. [27], [28] makes all test cases be of an observable sort by transforming each traditional test case $(\tau = \tau')$ into a set of test cases
Kong et al.’s solution to the problem was conditional test formal specification and requires a logic inference engine. This again assumes completeness for the conditions are true. This again assumes completeness for the

can be validated prior to testing by deciding whether the

Test Cases:
A test case in the form of a conditional equation

4) Validating Test Case: Logic Inference vs Conditional Test Cases: A test case in the form of a conditional equation can be validated prior to testing by deciding whether the conditions are true. This again assumes completeness for the formal specification and requires a logic inference engine. Kong et al.’s solution to the problem was conditional test cases to postpone the validation of test cases to runtime.

A conditional test case \( \langle \tau = \tau', i f \ c \rangle \) is a triple \( \langle \tau_1, \tau_2, c \rangle \), where \( \tau_1 \) and \( \tau_2 \) are ground terms, and the condition \( c \) is a set of checkable equations. To test this, execute the condition and if it is true, execute each of the ground terms and compare the results; if it is false, do nothing.

5) Execution of Test Cases: A test case must be translated into a sequence of invocations of operations on the software entity under test. Various techniques have been advanced for testing procedural programs [24], OO programs [25], [26] and software components [27], [28]. None of them can be applied immediately to web services because they all assume an ability to create and initialise arbitrary instances of the entity, or to copy and store the entity for comparison. Web services in general, however, can neither be reinitialised or copied since they may be third-party.

IV. THE PROPOSED APPROACH

We adopt a standard approach of instantiating conditional checkable positive test cases from axioms but with a few novelties added. We start with three notions.

- floating test cases, which have terms containing a variable that represents the state of the entity under test, can be applied without resetting or initialising that state and are in contrast to fixed test cases, which are formed entirely from ground terms;
- controllable sorts, which model entities whose state can be saved and then recovered, after the execution of a few operations; and
- monic test cases, which are those that require only one copy of entity under test since they execute a linear sequence of operations.

We prove that if a sort is controllable then its checkable test cases are monic, but if it is uncontrollable then the test cases may or may not be monic. We have devised an algorithm to decide this and to generate a monic execution sequence if one exists. Our case study shows that for one real industrial example of a web service, a significant proportion (about 20%) of the specification units are uncontrollable. This necessitates our new technique, which, also shown by the case study, succeeds in testing all but a few (less than 5%) of the service’s axioms.

In this section we define the three notions more formally and present the theorems underlying the algorithms for our technique. Proofs are omitted to save space and the algorithms themselves are given in [35], but summarised in Section V.

A. Floating Test Cases

Consider the following axiom of StackService.

\[
\forall s : \text{StackService}, x : \text{Int.}(s; \text{push}(x); \text{pop} = s)
\]  

(1)

Existing algebraic testing techniques would generate a test case by substituting ground terms for variables. For example, with 2 for \( x \), and \( \text{nil;push}(1) \) for \( s \), representing an initial starting state of a stack containing only the element 1, we obtain the following.

\[
\text{nil;push}(1); \text{push}(2); \text{pop} = \text{nil;push}(1)
\]  

(2)

To test this we need to initialise the entity. We can do this either by creating a new instance of the entity and setting it to a standard initial state, such as the empty stack here, or by resetting an already existing entity to that initial state.

The ability to perform initialisations is invaluable when testing the original authors of a web service but it will not be made available to its clients because each initialisation will be global and affect all users. So a client, when testing a web service before dynamic composition, must use the current state. If we call this \( s \), then the following is a suitable test case that pushes 2 onto the stack, pops it off and then checks that the stack is as it was before the push.

\[
s; \text{push}(2); \text{pop} = s
\]  

(3)

We call this a floating test case because it can be applied to the entity whatever its current state. As a test case, it is unusual for not using ground terms and in fact it must not have constants of the same sort as the entity. More formally, Definition 1: (Fixed and Floating Test Cases)

Given an algebraic specification \( S \), a test case \( T = \langle \tau_1, \tau_2, c \rangle \) for testing an entity of sort \( s \) is floating, if it contains only one variable, which must be of sort \( s \), and it does not contain any constants of sort \( s \). Test case \( T \) is fixed, if it does not contain any variables. □

Example 2: Equ (3) above is a floating test case derived from the axiom in Eqn (1) and so is Eqn (4) below.

\[
s; \text{push}(1); \text{push}(2); \text{pop} = s; \text{push}(1)
\]  

(4)

□

Note that for some axioms only fixed test cases can be derived. Examples include the axioms \( \text{nil.isEmpty} = \text{True} \) and \( \text{nil.length} = 0 \). In general, the following theorem can be used to determine whether an axiom has floating test cases.

Theorem 1: (Existence of Floating Test Cases)

An axiom has an instance that is a floating test case for sort \( s \), if the equation of the axiom has a universally quantified variable of sort \( s \) and it contains no constants of sort \( s \). □
In the sequel, we say an axiom is **floatable** for sort $S$ if it has an instance that is a floating test case for sort $S$. By Theorem 1, we can easily see that the axioms on Line 12 and 13 are not floatable but all the other axioms are. Similarly, a term for sort $S$ can also be said to be floatable if it contains only one variable of sort $S$ and no constant of sort $S$. Theorem 2 follows the definition of floatable test cases.

**Theorem 2**: (Construction of Floating Test Cases)
Let $Ax$ be a floatable axiom for sort $S$.

1) An instance $T$ of an axiom $Ax$ is a floating test case, if $T$ is obtained by substituting every variable of a sort $s' \neq s$ with a ground term and leaving one variable of sort $s$ as a free variable in the equation of the axiom.

2) $T'$ is a floating test case, if it is obtained from a floating test case $T$ by substituting a floatable term of sort $S$ in place of the free variable of sort $S$ in $T$.

We call test cases obtained by the two methods above **primary** and **derived** floating test cases, respectively.

**Example 3**: Here are some primary floating test cases based on the axioms of StackService.

\[
\begin{align*}
\text{s.isEmpty} &= \text{False}, \text{ if } \text{s.length} > 0 \quad (5) \\
\text{s.pop.length} &= \text{s.length} - 1, \text{ if } \text{s.length} > 0 \quad (6)
\end{align*}
\]

Here are some derived floating test cases based on them.

\[
\begin{align*}
\text{s.push(1).isEmpty} &= \text{False}, \text{ if } \text{s.push(1).length} > 0 \quad (7) \\
\text{s.push(1).pop.length} &= \text{s.push(1).length} - 1 \quad \text{if } \text{s.push(1).length} > 0 \quad (8)
\end{align*}
\]

Note that conditional test cases are not just convenient but necessary too because often the truth of a condition cannot be determined statically. Floating test cases can be checkable, just like fixed test cases are, if all of their terms are observable. They can be obtained by composing observation contexts, such as in the following examples, derived from Equ (4).

\[
\begin{align*}
x.\text{push(1).push(2).pop.length} &= x.\text{push(1).length}, \quad (9) \\
x.\text{push(1).push(2).pop.top} &= x.\text{push(1).top}, \quad (10) \\
x.\text{push(1).push(2).pop.pop.top} &= x.\text{push(1).pop.top} \quad (11)
\end{align*}
\]

**B. Controllable Sort**

A major difficulty in testing third-party web services is lack of controllability, a notion we now formally define. First note that a floatable term $\tau$ can be written $\tau(x)$ to indicate that $x$ is the variable that appears free in it.

**Definition 2**: (Controllable Sort) A sort $S$ in algebraic specification $S$ is **controllable** if for each floatable term $\tau(x)$ of sort $S$, there is a pair $\rho_x(\tau)$ and $\delta_x(\tau)$ of floatable terms for sort $S$ such that for all ground terms $\theta$ of sort $S$, we have that

\[
\begin{align*}
S \vdash \delta_x(\tau(\rho_x(\theta))) &= \theta, \\
S \vdash \tau(\rho_x(\theta)) &= \tau(\theta).
\end{align*}
\]

We call $\rho_x(x)$ and $\delta_x(x)$ the **recorder** and **recooker** of $\tau$ on sort $S$, and use $\text{Recorder}_S(\tau)$ and $\text{Recooker}_S(\tau)$ to denote them, respectively.

Informally, suppose that a sequence of operations $\tau$ is applied to an entity in state $\theta$, changing it to a new state $\theta' = \tau(\theta)$. If sort $S$ is controllable then applying $\rho_x$ to $\theta$ first, before applying the operations in $\tau$, will make it possible to recover $\theta$ afterwards by applying $\delta_x$. Both $\rho$ and $\delta$ are parameterised by $\tau$ since in general the knowledge of $\tau$ is required to define both the recording process and the corresponding recovery. One special case of a controllable sort is a recordable sort, where $\delta_x$ and $\rho_x$ are independent of $\tau$ so the subscript $\tau$ can be dropped from the conditions. Another special case is a reversible sort, where $\delta_x$ does still depend on $\tau$ but a recorder $\rho_x$ is not needed.

**Definition 3**: (Reversible and Recordable Sorts) A sort $S$ specified in algebraic specification $S$ is **reversable** if for every floatable term $\tau(x)$, there is a floatable term $\delta_x$ such that for all ground terms $\theta$ we have

\[
S \vdash \delta_x(\tau(\rho_x(\theta))) = \theta, \\
S \vdash \tau(\rho_x(\theta)) = \tau(\theta).
\]

We say that $\delta_x$ is the reverse of $\tau$, and write it as $\tau^{-1}$.

A sort $S$ specified in $S$ is **recordable**, if there are floatable terms $\rho(x)$ and $\delta(x)$ such that for all floatable terms $\tau(x)$ for sort $S$ and ground terms $\theta$ of sort $S$ we have that

\[
\begin{align*}
S \vdash \delta(\tau(\rho(\theta))) &= \theta, \\
S \vdash \tau(\rho(\theta)) &= \tau(\theta).
\end{align*}
\]

Once again, similar to controllable sorts, $\rho(x)$ is called the **recorder** for sort $S$ and written $\text{Recorder}_S$, and $\delta(x)$ is called the **recooker** for sort $S$ and written $\text{Recooker}_S$.

The sorts for primitive datatypes like $\text{Integer}$, $\text{Bool}$, $\text{Character}$ and $\text{String}$ are controllable because we can always copy the values of such entities to save them and then restore their values no matter how many operations have been performed on them in the meantime. Formally, for each floatable term $\tau(x)$ of a primitive sort $S$, where $\tau$ is a ground term, we can perform $(y := x); \tau(x); (x := y)$, where $y$ is an entity of the primitive sort $S$. The following theorem states that reversible and recordable are special cases of controllable.

**Theorem 3**: If a sort $S$ specified in algebraic specification $S$ is reversable then it is controllable. In addition, if it is recordable then it is controllable.

As the following example demonstrates with StackService, a sort can be controllable even if we cannot copy and save the whole state of the service.

**Example 4**: (Controllability of StackServices)

If $\tau(x)$ is a floatable term for sort StackService, then it can be transformed into one of the following two forms.
In the case of the first form, let
\[ \delta_r(x) = x; \text{push}(a_1); \text{push}(a_2); \ldots; \text{push}(a_k) \]  \hspace{1cm} (12)
\[ x; \text{pop}; \text{pop}; \ldots; \text{pop}. \]  \hspace{1cm} (13)

By induction on \( k \), we can prove the two controllability conditions \( \delta_r(\tau(\rho_r(x))) = x \) and \( \tau(\rho_r(x)) = \tau(x) \) by applying axioms of \textit{StackServices}. We can use induction again to prove these same two equations for the second form.

This time we let
\[
\delta_r(x) = x; \text{push}(a_1); \text{push}(a_2); \ldots; \text{push}(a_k); \ldots; \text{push}(a_1),
\]
\[ \rho_r(x) = x; \text{push}(a_k); \ldots; \text{push}(a_1) \]

The following theorem gives for any controllable sort the required execution sequences for checkable floating test cases.

**Theorem 4:** (Testing A Controllable Sort)

Let \( S \) be a given algebraic specification, \( s \) be a controllable sort, \( T = (\tau_1 = \tau_2, \text{if } c) \) be a checkable floating test case for sort \( s \), and \( E \) an instance of sort \( s \).

\( E \) is a correct implementation of specification \( S \) implies that the following execution of the test case \( T \) returns \textit{true}.

\begin{enumerate}
  \item Invocation sequence of \textit{Recorder}(c); (*save the state of \( E^* \))
  \item Invocation sequence of \textit{c}.
  \item Invocation sequence of \textit{Recoverer}(c); (*restore the state of \( E^* \))
  \item If result of \textit{c} is \textit{True} then
    \begin{enumerate}
      \item Invocation sequence of \textit{Recorder}(\tau_1); (*save the state of \( E^* \))
      \item Invocation sequence corresponding to \( \tau_1 \)
      \item Save the result of \( \tau_1 \) to local variable \( v_1 \)
      \item Invocation sequence of \textit{Recoverer}(\tau_2); (*restore state of \( E^* \))
      \item Invocation sequence of \( \tau_2 \)
      \item Save the results of \( \tau_2 \) to a local variable \( v_2 \)
      \item Check \( v_1 == v_2 \)
    \end{enumerate}
\end{enumerate}

\( \square \)

We now illustrate this for \textit{StackServices}. The following execution sequence below is for test case (9), where \( A \) denotes the service.

\begin{verbatim}
A.push(1);
Int RB = A.length;
A.pop;  (*Reverse to the original state*)
A.push(1);
A.push(2);
A.pop;
Int RA = A.length;
Assert(RA == RB);
\end{verbatim}

\section*{C. Monic Test Cases}

A key feature of the example test execution sequence we have just seen is that it requires only one instance of the entity under test so it can be used for testing web services. The question now is whether we can generate such an execution sequence when the sort is uncontrollable? We now define the concept of \textit{monic} test cases, which are those that can be implemented by a linear sequence of operation invocations applied to a single entity of the sort.

**Definition 4:** (Monic Execution)

Let \( T \) be a test case for a sort \( s \) specified in a specification \( S \). A \textit{monic execution} of \( T \) is a linear sequence of operations that only applies to one instance entity \( E \) of sort \( s \) such that if \( E \) is correct on test case \( T \) with respect to the specification \( S \) then the test execution returns \textit{true}.

A test case \( T \) is \textit{monic}, if it has a monic execution. An axiom is \textit{monic}, if it has at least one instance that is a monic test case. \( \square \)

Therefore, by Definition 4 and Theorem 4, we have the following theorem.

**Theorem 5:** For controllable sorts, all checkable floating test cases are monic. \( \square \)

However, for uncontrollable sorts, test cases and axioms can be non-monic and even a term can be, as shown in this example that specifies a sort for lists of numbers.

\begin{verbatim}
Example 5: (Non-monic term)
1: Spec List uses Integer;
2: Operation
3:  Sorting: List -> List;
4:  */ Sort the list*/
5:  Merge: List, List -> List;
6:  */merge 2 sorted lists into 1 sorted list*/
7:  Odd: List -> List;
8:  */get the odd elements of the list*/
9:  Even: List -> List;
10: */get the even elements of the list*/
11:  Axiom ... End
12:  End
\end{verbatim}

\begin{verbatim}
The following term cannot be implemented correctly by a linear sequence of invocations on a single entity of \textit{List}.
\begin{verbatim}
Merge(s.Odd.Sorting, s.Even.Sorting).
\end{verbatim}
\end{verbatim}

\( \square \)

In the sequel, we say that a term \( \tau(x) \) is monic if it can be correctly implemented by a monic execution. The following theorem gives some properties of monic terms and test cases.

**Theorem 6:** (Monic Terms and Test Cases)

Let \( x \) be a variable of sort \( s \); \( \varphi \) be an operation on \( s \); \( \alpha \) and \( \alpha_i \) be attributes of sort \( s \); \( \vec{a}, \vec{b} \) and \( \vec{b}' \) be floatable terms or ground terms that do not change the state of \( x \); and \( c \) be a condition that does not change the state of \( x \).

\begin{enumerate}
  \item A floatable term \( \tau(x) \) in one of the forms below is monic.
    \begin{verbatim}
    x; \varphi(\vec{a}),  x.\alpha(\vec{a})
    \end{verbatim}
    \hspace{1cm} (15)
  \item If floatable terms \( \tau_1(x) \) and \( \tau_2(x) \) are monic, \( \tau_1(\tau_2(x)) \) is also monic.
  \item A test case for sort \( s \) in the form below is monic.
    \begin{verbatim}
    x; \varphi(\vec{a}), \alpha(\vec{b}) = Op(x.\alpha_1(\vec{b}_1), \ldots, x.\alpha_n(\vec{b}_n)), \text{if } c.
    \end{verbatim}
    \hspace{1cm} (16)
\end{enumerate}

\( \square \)

It is worth noting that the majority of axioms that we have come across in the specification of services and software components are in the format of (16).
Our test case generation process consists of three main steps:

1. For each axiom to be tested, instantiate it to produce a set of primary floating test cases;
2. For each of these floating test cases, if the sort is not observable, generate a set of checkable test cases by composing them with applicable observation contexts;
3. For each of those test cases, if it is monic, generate a monic execution sequence.

Step 1 and 2 are based on the theorems given in the previous section so we will focus here on Step 3.

A. Test Execution Graph

To determine whether a test case is monic and to generate an execution sequence if it is, we construct a directed graph to model the state transitions that the test execution will cause. We therefore call the graph a test execution graph (TEG).

**Definition 5:** (Test Execution Graph)

A test execution graph (TEG) for a set of terms \( \{\tau_1, \ldots, \tau_k\} \) is a finite directed graph \( G(s) = (N, A) \), where \( N \) and \( A \) are finite sets of nodes and arcs, respectively, such that

1. \( N \) is partitioned into entity nodes \( N^e \) and operation nodes \( N^o \); i.e. \( N = N^e \cup N^o \) and \( N^e \cap N^o = \emptyset \). These will be drawn as circles and rectangles respectively.
2. Each node \( n \) in \( N^e \) is labelled \( x : s \) and represents the state of an entity \( x \) of sort \( s \), where \( x \) can be either a variable or a constant and \( s \) must be one of the sorts of the specification. The \( x \) can be omitted for nameless entities that are created at run-time.
3. Each node \( n \) in \( N^o \) is labelled with an operator or an attribute.
4. \( A \subseteq (N^e \times N^o) \cup (N^o \times N^e) \). An arc \( a = (n, n') \in A \) represents a data flow from node \( n \) to \( n' \) and is graphically depicted with an arrow from \( n \) to \( n' \).

**B. Generation of TEG**

Given a checkable floating test case in the form

\[ \tau_0 = \tau_0', \text{ if } \tau_1 = \tau_1', \ldots, \tau_n = \tau_n', \]

the first step is to generate TEGs for the terms \( \tau_i \) and \( \tau_i' \), \( i = 0, \ldots, n \). The algorithm for this is given in Appendix A of [35].

The second step is connecting these TEGs together by adding operation nodes labeled with “==” for checking the equations and entity nodes of Boolean type to store the results. The conditions of the test case are then connected to the equation by adding an operation node IsTrue between the Boolean nodes and the start nodes of the equations. The algorithm for this is given in Appendix B of [35].

The single connected TEG for a test case is then reduced by repeatedly performing merges until there are no more applicable merges. The four types of merges are as follows:

1. any two root entity nodes with the same label;
2. any two operation nodes with the same label if their entry arcs come from the same entity nodes;
3. any two entity nodes with the same label from the same operation;
4. any two arcs with the same source node and the same target node.

**Example 6:** Consider a sort RepStack, like StackService but with an additional operator replace(Int) that replaces the element on top of the stack with the parameter supplied. Consider the following test case.

\[ x; push(1); replace(2).top = x; push(2).top, \text{ if } x.length > 1. \]

As shown in Fig. 1, three TEGs are constructed: one of each side of the equation, and one for the condition. They are then connected by the nodes and edges in blue ink. After the merging step we finish with the TEG shown in Fig. 2.

---

**Figure 1. Test Execution Graphs Generated from Terms and Connected**

**Figure 2. Test Execution Graph Obtained by Merging Common Nodes**

Once we have generated a TEG for a test case, we can determine whether it is monic using the theorem below.

**Theorem 7:** (Existence of Monic Test Execution)
Let \( G \) be a TEG generated from a checkable floating test case \( T \). Then \( T \) is not monic, if and only if there exists an entity node \( d \) in \( G \) such that

1) \( d \) is labeled with a sort \( s \) that is not controllable, and
2) \( d \) links to at least two different operation nodes \( op_1 \) and \( op_2 \), each linked to a different entity node labelled with sort \( s \). □

For example, \( \text{RepStack} \) is no longer controllable due to the extra operation \( \text{replace} \). By Theorem 7, we cannot generate a monic execution of the test case in Example 6. This is because in the TEG shown in Fig. 2 the start node labeled with \( x : \text{RepStack} \) is linked to two operation nodes that lead to two different entity nodes labeled with \( x : \text{RepStack} \).

C. Generation of Monic Test Sequence

Given a connected and merged TEG for a checkable floating test case \( T \), the test execution sequence can be generated with the algorithm given in Appendix C of [35], which generates sequences based on cases for entity nodes and operation nodes, according to the following two rules.

**Rule 1:** For a subset of paths in the form of pattern A shown in Figure 3, we generate this:

\[
O_{p_1,1}(x_1); O_{p_1,2}(x_1); \ldots; O_{p_1,n_1}(x_1); \\
O_{p_2,1}(x_2); O_{p_2,2}(x_2); \ldots; O_{p_2,n_2}(x_2); \\
\ldots \\
O_{p_k,1}(x_k); O_{p_k,2}(x_k); \ldots; O_{p_k,n_k}(x_k); \\
O(x_1, x_2, \ldots, x_k);
\]

For case (b), we must make sure the operation node does not produce two or more entity nodes of sort \( s \). If it does, the algorithm must terminate with the verdict “not monic”. If not, call the operation node \( O_{p_k,1} \) and generate this:

\[
O_{p_1,1}(x_1); O_{p_1,2}(x_1); \ldots; O_{p_1,n_1}(x_1); \\
O_{p_2,1}(x_2); O_{p_2,2}(x_2); \ldots; O_{p_2,n_2}(x_2); \\
\ldots \\
O_{p_k,1}(x_k); O_{p_k,2}(x_k); \ldots; O_{p_k,n_k}(x_k);
\]

□

![Figure 3. Pattern A of Paths in Test Execution Graph](image)

**Rule 2:** For a subset of paths in the form of pattern B shown in Figure 4, a monic execution sequence can be generated if (a) the sort \( s \) is controllable; or (b) only one of the operation nodes produces an entity node of sort \( s \).

For case (a), we generate this:

\[
\text{Var } y : s; \\
y := x; O_{p_1,1}(x); O_{p_1,2}(x_1); \ldots; O_{p_1,n_1}(x_1); \\
x := y; O_{p_2,1}(x); O_{p_2,2}(x_2); \ldots; O_{p_2,n_2}(x_2); \\
\ldots \\
x := y; O_{p_k,1}(x); O_{p_k,2}(x_k); \ldots; O_{p_k,n_k}(x_k);
\]

For case (b), we must make sure the operation node does not produce two or more entity nodes of sort \( s \). If it does, the algorithm must terminate with the verdict “not monic”.

![Figure 4. Pattern B of Paths in Test Execution Graph](image)

VI. Case Study

We have implemented the test case generation technique outlined above in an automated testing tool and used it in a case study, designed to demonstrate that it is applicable to real industrial web services. The RESTful web service we chose was GoGrid as we had already developed an algebraic specification for it in previous work [33], [34].

GoGrid is the world’s largest pure-play Infrastructure-as-a-Service provider specialising in Cloud Infrastructure solutions. It provides an API with which its customers can easily and dynamically deploy and manage their applications and workloads through a programmatic interface.

The GoGrid API has a REST-like query interface and it has 5 types of common operations: List, Get, Add, Delete and Edit. These types of operations can be applied to 8 different types of objects: Job, Load balancer, Server, Image, IP, Passwords, Billing and Utility Option. Some of the operations are not applicable to all types of objects, while some objects have additional special operators. Table I gives the applicable operators for each type of objects.

The specification of GoGrid is based on a framework for specifying RESTful web services. That framework defines the common structure and features of all RESTful web services. Examples of this include the structure of HTTP requests and responses in a set of specification units in SOFIA. A concrete RESTful web service can be specified in a number of specification units that extend the framework units. The whole GoGrid API has been specified in SOFIA. Table II gives the numbers of specification units in GoGrid.
specification. The column #Obs gives the number of specification units that are observable and the column #Ctrl gives the number of specification units that are controllable. The data show that there are a significant proportion (20.2%) of entities in GoGrid specification that are not controllable. Moreover, these entities are the GoGrid objects, and therefore the most important entities of the web service, so monic test execution sequences must be generated for them.

Among those entities that are not controllable, 96.6% of the axioms are monic as shown in Table III, and 92.5% of all axioms are floating as shown in Table II, where the column #Axiom gives the total number of axioms in the type of units and the column #FAx gives the number of floatable axioms.

Therefore, our proposed technique is applicable to a large proportion of axioms in the specification of web services.

VII. Conclusion and Future Work

In this paper, we proposed an approach for generating test cases for web services using algebraic specifications written in SOFIA. A case study with a real industrial service demonstrated the feasibility of the proposed approach.

We are now conducting more experiments to evaluate the fault detection ability of the monic testing technique. An observation that we have made in the case study is that, although there are some axioms that are not monic, these axioms can be rewritten into one or more equivalent axioms and thereby become monic. We are further studying how to transform non-monic axioms into equivalent monic axioms.

ACKNOWLEDGEMENT

The work reported in this paper is partially supported by National Natural Science Foundation of China (Grant No. 61502233) and Jiangsu Qinglan Project, and partially supported by EU FP7 project MONICA on Mobile Cloud Computing (Grant No.: PIRSES-GA-2011-295222).

REFERENCES
