Algebraic Testing of Web Services: The Monic Approach

Hong Zhu, Ian Bayley
Dept of Comp. and Comm. Technologies
Oxford Brookes University
Oxford OX33 1HX, UK
hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

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

***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 first introduction in the early 1980s [23], the algebraic testing technique has been advanced significantly. The most well know 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 CASOCC 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.

$$deposit(acnt, \$200); withdraw(acnt, \$100); balance(acnt) = deposit(acnt, \$100); balance(acnt)$$

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 Concepts of Algebraic Specifications: An algebraic specification consists of a *signature* Σ and a set Ax of *axioms*. The signature comprises a set S of *sorts* and a set Φ 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 τ and τ' 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 $(\mathbf{S}, \mathbf{\Sigma}, \mathbf{Ax})$, where

1) $\mathbf{S} = \langle S, \succ, \triangleright \rangle$, where S is a finite set of sorts, $\succ$ and $\triangleright$ 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) $\mathbf{\Sigma} = \{\Sigma_s | s \in S\}$ is a set of signatures indexed by s, where Σ_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) $\mathbf{Ax} = \{Ax_s | s \in S\}$ is a finite set of axiom sets indexed by s, where the semantics of the operators in Σ_s are defined by the set $Ax_s \cup \bigcup_{s \triangleright x} Ax_x$

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

```
<Spec unit> ::=
  Spec <Sort Name> [<Observability>];
    [extends <Sort Names>] [uses <Sort Names>]
  <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 Σ_s. It is a constant if $n = 0$, $m = 1$, and $s'_1 = s$, but an attribute if $n > 0$, $s \in \{s_1, \cdots, s_n\}$, and $s \notin \{s'_1, \cdots, s'_m\}$.

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
```

□

Note that signature declarations for operators resemble those of methods in OO programming languages, in that $push(Integer) : Message$ is equivalent to this:

$$push : StackService, Integer \rightarrow StackService, Message.$$

Similarly, terms in SOFIA can be written in a so-called *dot format* to resemble the expressions of OO programming languages. For example, the following two conditional equations are equivalent.

$$s.isEmpty = False,\ if\ s.length > 0$$
$$isEmpty(s) = False,\ if\ length(s) > 0$$

Moreover, we introduce a further format, the *;-format* which enables us to distinguish the state of a service after an operation and the response message of the service request. Let e be a term representing an entity of sort s. Then $e.[\varphi(\vec{x})]$ represents the state of e after φ has been performed with parameters $\vec{x}$, and $e.\varphi(\vec{x})$ represents the reply message. As a syntactic sugar, we write $\tau; \varphi_1(\vec{a}_1); \cdots ; \varphi_k(\vec{a}_k)$ to denote $\tau.[\varphi_1(\vec{a}_1)].\cdots .[\varphi_k(\vec{a}_k)]$. For example, $s; push(x); pop$ denotes $s.[push(x)].[pop]$.

Informally, a sort is observable if it has an operation $==$ defined on it that can be used to tell if two terms of the sort are equal. The primitive sorts of SOFIA i.e. $Integer$, $Boolean$, $Character$, and $String$ are all observable but many structured datatypes, software components and services cannot be checked for equality in this way. Any user-defined sort that is observable is declared to be so, with an indicator of which operator plays the role of $==$. Defining observability formally, for an algebraic specification S, a sort s is *observable*, if there is a binary predicate $==$ defined on s such that for all ground terms τ and τ' of sort s, $S \vdash \tau = \tau'$ if and only if $S \vdash (\tau == \tau') = true$.

B. Algebraic Testing

Algebraic testing is based on the observation that each ground term of a signature can be interpretted 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 $\langle \tau, \tau' \rangle$ of ground terms is *positive* if τ and τ' 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 to guarantee 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 τ and its normal form τ' 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 $\langle \tau = \tau' \rangle$ when the sort of τ and τ' 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 $\langle \tau = \tau' \rangle$ into a set of test cases $\{\langle \tau.\theta = \tau'.\theta \rangle \,|\theta(x) \in OC(x)\}$, where $OC(x)$ is the set of observation contexts of sort s.

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 [27], [28].

A conditional test case $\langle \tau = \tau', if\ c \rangle$ is a triple $\langle \tau_1, \tau_2, c \rangle$, where τ_1 and τ_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 web services, 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 the Appendices, but summarised in Section V.

A. Floating Test Cases

Consider the following axiom of $StackService$.

$$\forall s : StackService, x : Int.(s; push(x); pop = s) \quad (1)$$

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

$$nil; push(1); push(2); pop = nil; push(1) \quad (2)$$

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 for 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.

$$s; push(2); pop = s \quad (3)$$

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.

$$s; push(1); push(2); pop = s; push(1) \quad (4)$$

□

Note that for some axioms only fixed test cases can be derived. Examples include the axioms $nil.isEmpty = True$ and $nil.length = 0$. 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 floating 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 $StackService$.

$$s.isEmpty = False,\ if\ s.length > 0 \tag{5}$$
$$s; pop.length = s.length - 1,\ if\ s.length > 0 \tag{6}$$

Here are some derived floating test cases based on them.

$$s; push(1).isEmpty = False,\quad if\ s; push(1).length > 0 \tag{7}$$
$$s; push(1); pop.length = s; push(1).length - 1$$
$$if\ s; push(1).length > 0 \tag{8}$$

□

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).

$$x; push(1); push(2); pop.length = x; push(1).length, \tag{9}$$
$$x; push(1); push(2); pop.top = x; push(1).top, \tag{10}$$
$$x; push(1); push(2); pop; pop.top = x; push(1); pop.top \tag{11}$$

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 τ 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 $\rho_\tau(x)$ and $\delta_\tau(x)$ of floatable terms for sort s such that for all ground terms θ of sort s, we have that

$$S \vdash \delta_\tau(\tau(\rho_\tau(\theta))) = \theta,$$
$$S \vdash \tau(\rho_\tau(\theta)) = \tau(\theta).$$

We call $\rho_\tau(x)$ and $\delta_\tau(x)$ the *recorder* and *recoverer* of τ on sort s, and use $Recorder_s(\tau)$ and $Recoverer_s(\tau)$ to denote them, respectively. □

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

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

$$S \vdash \delta_\tau\tau(\theta) = \theta.$$

We say that δ_τ is the *reverse* of τ, and write it as τ^{-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 θ of sort s we have that

$$S \vdash \delta(\tau(\rho(\theta))) = \theta,$$
$$S \vdash \tau(\rho(\theta)) = \tau(\theta).$$

Once again, similar to controllable sorts, $\rho(x)$ is called the *recorder* for sort s and written $Recorder_s$, and $\delta(x)$ is called the *recoverer* for sort s and written $Recoverer_s$. □

The sorts for primitive datatypes like $Integer$, $Bool$, $Character$ and $String$ 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 τ 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 reversible 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 Stack Services)

If $\tau(x)$ is a floatable term for sort $StackService$, then it can be transformed into one of the following two forms.

$$x; push(a_1); push(a_2); \cdots; push(a_k) \tag{12}$$
$$x; \underbrace{pop; pop; \cdots; pop}_{k}. \tag{13}$$

In the case of the first form, let

$$\delta_\tau(x) = x; \underbrace{pop; pop; \cdots; pop}_{k},\quad and\quad \rho_\tau(x) = x.$$

By induction on k, we can prove the two controllability conditions $\delta_\tau(\tau(\rho_\tau(x))) = x$ and $\tau(\rho_\tau(x)) = \tau(x)$ by applying axioms of $StackServices$. We can use induction

again to prove these same two equations for the second form. This time we let

$$\rho_\tau(x) = x; (a_1 := top); pop; (a_2 := top); pop; \\ \cdots ; (a_k := top); pop; push(a_k); \cdots ; push(a_1), \\ \delta_\tau(x) = x; push(a_k); \cdots ; push(a_1) \quad \square$$

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, $T = \langle \tau_1 = \tau_2,\ if\ c \rangle$ 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 *true*.

Begin

1) Invocation sequence of $Recorder(c)$; (*save the state of E*);
2) Invocation sequence of c;
3) Invocation sequence of $Recoverer(c)$; (*restore the state of E*)
4) If result of c is $True$ then
   a) Invocation sequence of $Recorder(\tau_1)$; (*save the state of E*)
   b) Invocation sequence corresponding to τ_1;
   c) Save the result of τ_1 to local variable v_1;
   d) Invocation sequence of $Recoverer(\tau_1)$; (*restore state of E*)
   e) Invocation sequence of τ_2;
   f) Save the results of τ_2 to a local variable v_2;
   g) Check $v_1 == v_2$.

End

□

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

```
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);
```

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 *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 *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 *True*.

A test case T is *monic*, if it has a monic execution. An axiom is *monic*, if it has at least one instance that is a monic test case.□

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

Theorem 5: For controllable sorts, all checkable floating test cases are monic. □

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.

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
```

The following term cannot be implemented correctly by a linear sequence of invocations on a single entity of $List$.

$$Merge(s.Odd.Sorting, s.Even.Sorting). \tag{14}$$

□

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; φ be an operation on s; α and α_i be attributes of sort s; $\vec{a}$, $\vec{b}$, and $\vec{b_i'}$ 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.

1) A floatable term $\tau(x)$ in one of the forms below is monic.
$$x; \varphi(\vec{a}), \quad x.\alpha(\vec{a}) \tag{15}$$
2) If floatable terms $\tau_1(x)$ and $\tau_2(x)$ are monic, $\tau_1(\tau_2(x))$ is also monic.
3) A test case for sort s in the form below is monic.
$$x; \varphi(\vec{a}).\alpha(\vec{b}) = Op(x.\alpha_1(\vec{b_1}), \cdots, x.\alpha_n(\vec{b_n})),\ if\ c. \tag{16}$$

□

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).

V. Algorithms

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 $\{\tau_1, \cdots \tau_k\}$ 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

$$\tau_0 = \tau_0',\ if\ \tau_1 = \tau_1', \cdots, \tau_n = \tau_n',$$

the first step is to generate TEGs for the terms τ_i and τ_i', $i = 0, \cdots, n$. The algorithm for this is given in Appendix A.

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.

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.

$$x; push(1); replace(2).top = x; push(2).top,\ \ if\ x.length > 1.$$

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. □

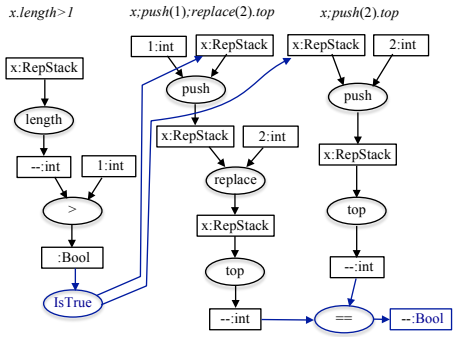

Figure 1. Test Execution Graphs Generated from Terms and Connected

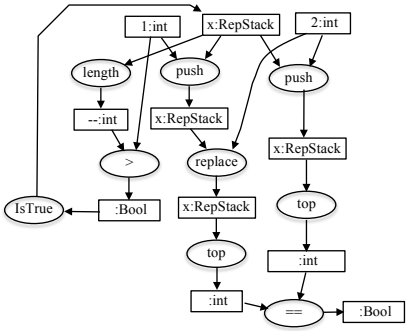

Figure 2. Test Execution Graph Obtained by Merging Common Nodes

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, $RepStack$ is no longer controllable due to the extra operation $replace$. 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 $x : RepStack$ is linked to two operation nodes that lead to two different entity nodes labeled with $x : RepStack$.

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, which generates sequences based on cases for entity nodes and operation nodes, according to these two rules.

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

$$
\begin{array}{l}
Op_{1,1}(x_1); Op_{1,2}(x_1); \cdots ; Op_{1,n_1}(x_1); \\
Op_{2,1}(x_2); Op_{2,2}(x_2); \cdots ; Op_{2,n_2}(x_2); \\
\cdots \\
Op_{k,1}(x_k); Op_{k,2}(x_k); \cdots ; Op_{k,n_k}(x_k); \\
Op(x_1, x_2, \cdots, x_k);
\end{array}
$$

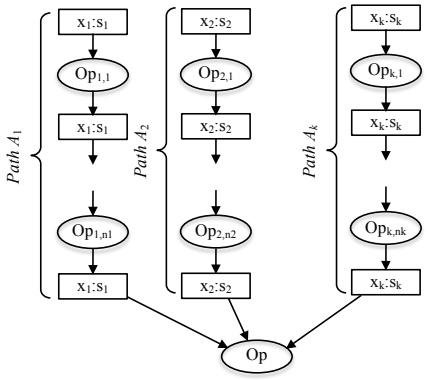

Figure 3. Pattern A of Paths in Test Execution Graph

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:

$$
\begin{array}{l}
Var\ y : s; \\
y := x; Op_{1,1}(x); Op_{1,2}(x_1); \cdots ; Op_{1,n_1}(x_1); \\
x := y; Op_{2,1}(x); Op_{2,2}(x_2); \cdots ; Op_{2,n_2}(x_2); \\
\cdots \\
x := y; Op_{k,1}(x); Op_{k,2}(x_k); \cdots ; Op_{k,n_k}(x_k);
\end{array}
$$

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 $Op_{k,1}$ and generate this:

$$
\begin{array}{l}
Op_{1,1}(x); Op_{1,2}(x_1); \cdots ; Op_{1,n_1}(x_1); \\
Op_{2,1}(x); Op_{2,2}(x_2); \cdots ; Op_{2,n_2}(x_2); \\
\cdots \\
Op_{k,1}(x); Op_{k,2}(x_k); \cdots ; Op_{k,n_k}(x_k);
\end{array}
$$

□

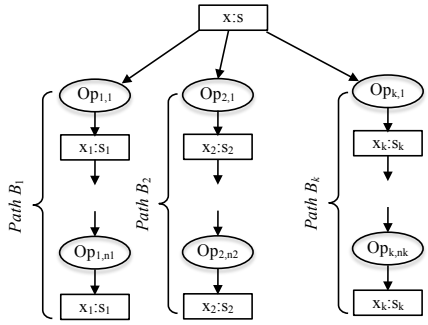

Figure 4. Pattern B of Paths in Test Execution Graph

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.

Table I
Applicable Operators on Objects

Object	List	Get	Add	Delete	Edit	Other
Load Balancer	Yes	Yes	Yes	Yes	Yes	
Server	Yes	Yes	Yes	Yes	Yes	Power
Server image	Yes	Yes		Yes	Yes	Save, Restore
Job	Yes	Yes				
IP	Yes					
Password	Yes	Yes				
Billing		Yes				
Option	Yes					

Table II
NUMBER OF UNITS AND AXIOMS IN GOGRID SPECIFICATION

Type of Spec Unit	#Units	#Obs	#Ctrl	#Axm	#FAx
WS Framework	10	10	10	11	4
Message Structures	12	12	12	17	17
Object Structures	11	0	0	8	8
Server Ops	13	12	12	29	29
Server Image Ops	13	12	12	30	30
Load Balancer Ops	11	10	10	28	28
Job Ops	5	5	5	9	9
IP Ops	3	2	2	3	2
Password Ops	5	4	4	5	4
Bill Ops	3	2	2	3	2
Option Ops	3	2	2	3	2
Total	**89**	**71**	**71**	**146**	**135**

Table III
STATISTICS OF MONIC AXIOMS OF UNCONTROLLABLE ENTITIES

Entity	#Axioms	#Monic
Server	14	13
Server Image	11	11
Load Balancer	13	12
Job	6	6
IP	4	4
Password	6	6
Bill	1	1
Option	4	4
Total	**59**	**57**

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

[1] Zhu, H., Zhang, Y., "Collaborative Testing of Web Services," IEEE Transactions on Services Computing, vol. 5, no. 1, pp. 116–130, 2012.

[2] 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.

[3] 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.

[4] 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 Matematicas e de Computacao, Universidade de Sao Paulo (USP), Sao Carlos, SP, Brazil. ISSN - 0103-2569, March 2010.

[5] Liu, D., Zhu, H., and Bayley, I., " From Algebraic Formal Specification to Ontological Description of Service Semantics", in *Proc. of IEEE ICWS 2013*, Santa Clara Marriott, CA, USA., pp. 579–586 ,2013.

[6] 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.

[7] 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.

[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 Berlin Heidelberg, 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 Berlin Heidelberg, 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 Berlin Heidelberg, 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 Berlin Heidelberg, pp. 143–157, 2009.

[12] Zhu, H. and He, X., "A methodology of testing high-level Petri nets", *Information and Software Technology*, Vol. 44, No. 8, June 2002, pp. 473–489.

[13] Xu, D., "A Tool for Automated Test Code Generation from High-Level Petri Nets", in *Proc. of PETRI NETS 2011*, Kristensen L.M. and Petrucci, L. (Eds.), LNCS 6709, pp. 308317, 2011.

[14] Zhang, X., Wu, C. and Xue, S. "Petri nets based test case selection model for service composition in cloud", in *Proc. of the IEEE DMA 2013*, pp. 914–917, 2013.

[15] 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.

[16] 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.

[17] Liu, D., Zhu, H., and Bayley, I., "Reference manual of the SOFIA algebraic specification language", Technical Report TR-CCT-AFM-01-2013, Department of Computing and Communication Technologies, Oxford Brookes University, Oxford, UK, 2013.

[18] ——, "SOFIA: An Algebraic Specification Language for Developing Services," in *Proc. of IEEE SOSE 2014*. Oxford, UK, pp.70–75, 2014.

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

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Hughes M., Stotts, D., "Daistish: systematic algebraic testing for OO programs in the presence of side-effects", in *Proc. ISSTA' 96*, ACM Press, pp, 53–61, 1996.

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

[28] Kong, L., Zhu, H., and Zhou, B., "Automated testing EJB components based on algebraic specifications," in *Proc. of IEEE COMPSAC'07*, vol. 2. Beijing, China, pp. 717–722, 2007.

[29] 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.

[30] 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.

[31] 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, Nov., 2013 .

[32] Zhu, H., "A Note on Test Oracles and Semantics of Algebraic Specifications", in *Proc. of QSIC'03*, Dallas,USA, pp. 91–99, 2003.

[33] 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.

[34] ——, "A case study on algebraic specification of cloud computing," in *Proc. of PDP 2013*, Belfast, Northern Ireland, pp. 269–273, 2013.

APPENDIX A. ALGORITHM TO GENERATE TEG

Algorithm 1 generates a TEG for a term. Note that, a term τ must be in the following form.

$$E; \varphi_1(\vec{a}_1); \cdots; \varphi_n(\vec{a}_n).\alpha(\vec{q}) \otimes_1 \vartheta_1(\vec{b}_1) \otimes_2 \cdots \otimes_m \vartheta_m(\vec{b}_m),$$

where $n, m \geq 0$; E is either a constant C or a variable x of sort s; φ_i, $i = 1, \cdots, n$, are operations or attributes of sort s; α is an attribute of sort s; ϑ_i, $i = 1, \cdots, m$, are attributes or operations of an imported sort $s'_i \neq s$; for each $i = 1, \cdots, m$, $\otimes_i$ is either ";" or "."; $\vec{a}_i$, $\vec{b}_i$ and $\vec{q}$ are sequences of terms in the above form.

Algorithm 1 TEG(τ) : Construction of TEG from Term τ

Input: A term τ (* τ must be in the form of (**??**) *).
Output: $\Gamma = (N^e, N^o, Ar)$, where $(N^e \cup N^o, Ar)$ is the TEG.
Step 1: (* Initialize *)
$N^e = \emptyset$; $N^o = \emptyset$; $Ar = \emptyset$;
Step 2: (* Process the main term *)
2.1. (*Construct nodes and edges *)
$N^e := \{d_0, \cdots, d_{n+m+1}\}$;
(* d_0 and d_{n+m+1} are called the *start* and *end* entity nodes *)
$N^o := \{d'_1, \cdots, d'_{n+m+1}\}$;
$Ar := \{e_1, \cdots, e_{n+m+1}\} \cup \{e'_1, \cdots, e'_{n+m+1}\}$, where
$e_i = \langle d_{i-1}, d'_i \rangle$, for $i = 1, \cdots, n+m+1$,
$e'_i = \langle d'_i, d_i \rangle$, for $i = 1, \cdots, n+m+1$;
2.2. (* Label entity nodes *)
if $E = x$ (* where x is a state variable of sort s *) **then**
Label entity node d_0 with "x:s";
for $i = 1, \cdots, n$ **do**
label d_i with "x:s"
end for
else (* $E = C$, where C is a constant of sort s *)
Label entity node d_0 with "C:s";
for $i = 1, \cdots, n$ **do**
label d_i with "C_v:s" (* where C_v is a new variable *)
end for
end if
Label entity node d_{n+1} with "–:s'; (*where s' is the result sort of attribute α *)
for $i = n+1, \cdots, n+m-1$ **do**
label entity node d_{1+i} with "–:s_i" (* where s_i is the result sort of ϑ_i *)
end for
Label entity node d_{n+m+1} with "–:s_m" (*where s_m is the result sort of ϑ_m *)
2.3. (* Label operation nodes *)
for $i = 1, \cdots, n$ **do**
label operation nodes d'_i with φ_i;
end for
Label operation node d'_{n+1} with α;
for $i = n+2, \cdots, n+m+1$ **do** label d'_i with ϑ_i;
end for
Step 3: (* Process sub-terms *)
for $\tau_t \in \{\vec{a}_i, \vec{b}_i, \vec{q}\}$ **do**
$\Gamma^\tau_t := TEG(\tau_t)$;
$N^e := N^e \cup \Gamma^\tau_t.N^e$;
$N^o := N^o \cup \Gamma^\tau_t.N^o$;
$Ar := Ar \cup \Gamma^\tau_t.Ar \cup \{\langle d^a, d^b \rangle\}$ (* d^a is the end entity node of term τ_t, d^b is the operation node of φ_i, α, or ϑ_i corresponding to subterm τ_t. *)
end for
return Γ.

APPENDIX B. ALGORITHM FOR CONNECTING TEGS

Algorithm 2 connects the TEGs of the terms in one test case into one TEG by adding operator nodes labelled with "==" and $IsTrue$ and arcs to link them together.

Algorithm 2 ConnectTEGs: Connect TEGs of a Test Case

Input: TEGs $(G_0, \cdots, G_n, G'_0, \cdots, G'_n)$.
Output: (N^e, N^o, Ar), where $(N^e \cup N^o, Ar)$ is the TEG.
Step 1. (*Initialisation *)
$N^e = \bigcup_{i=0}^n G_i.N^e \cup \bigcup_{i=0}^n G'_i.N^e$;
$N^o = \bigcup_{i=0}^n G_i.N^o \cup \bigcup_{i=0}^n G'_i.N^o$;
$Ar = \bigcup_{i=0}^n G_i.Ar \cup \bigcup_{i=0}^n G'_i.Ar$;
Step 2. (* Connect Terms for Each Equation *)
for $i = 0, 1, \cdots, n$ **do**
$N^o := N^o \cup \{c'_i\}$, and label c'_i with operation $==$;
$N^e := N^e \cup \{c_i\}$ and label c_i with label "–: Bool";
$Ar := Ar \cup \{\langle c'_i, c_i \rangle\}$;
$Ar := Ar \cup \{\langle de_i, c'_i \rangle\}$, where de_i is the end entity node of G_i.
$Ar := Ar \cup \{\langle de'_i, c'_i \rangle\}$, where de'_i is the end entity node of G'_i.
end for
Step 3. (* Connect Conditions to Equation)
$N^o := N^o \cup \{c''\}$ and label c'' with operation $IsTrue$;
$Ar := Ar \cup \{\langle c'', ds_0 \rangle\}$, where ds_0 is the start entity node of G_0;
$Ar := Ar \cup \{\langle c'', ds'_0 \rangle\}$, where ds'_0 is the start entity node of G'_0.
for $i = 1, \cdots, n$ **do**
$Ar := Ar \cup \{\langle c_i, c'' \rangle\}$;
end for

APPENDIX C: GENERATE MONOLITHIC CODE

This algorithm is for generating a test execution sequence from a TEG.

The algorithm uses two queues QE and QOp to recognise the path structures and control the processing of the nodes in the TEG. QE is a list of entity nodes to be processed. It initially contains all start nodes of the conditions and the entity nodes that have no inward arrows, which are constants or the state variable node. QOp is a set of operations to be processed. Its elements are in the form of $\langle Op, Rs, NRs \rangle$, where Op is an operator node, Rs and NRs are two lists of entity nodes representing those operants of Op that have already processed and not yet processed, respectively.

Table IV lists the functions used in the algorithm.

Table IV
FUNCTIONS USED IN ALGORITHM 3

Function	Meaning
$Get(Q : \text{List of } N^e)$	Return the top element of Q and remove it from Q.
$InDegree(x : N)$	The in degree of x
$OutDegree(x : N)$	The out degree of x
$NextOps(x : N^e)$	$\{y \in N^o \mid \langle x, y \rangle \in Ar\}$
$NextEs(x : N^e)$	$\{y \in N^e \mid \exists z \in N^o.(\langle x, z \rangle, \langle z, y \rangle \in Ar)\}$
$OpOutEs(x : N^o)$	$\{y \in N^e \mid \langle x, y \rangle \in Ar\}$
$OpInEs(x : N^o)$	$\{y \in N^e \mid \langle y, x \rangle \in Ar\}$
$Sort(x : N^e)$	The sort labeled on x
$Object(x : N^e)$	The object name labeled on x
$GenCode(x : String)$	Add string x to the end of $CodeSeq$
$NewVar(x : Sort)$	A new variable identifier of sort x

Algorithm 3 GenExecSeq(G): Generate test execution sequence from a TEG

Input: (N^e, N^o, Ar) (* $(N^e \cup N^o, Ar)$ *is the TEG for a test case t.* *)
Output: $CodeSeq$: *a monolithic sequence of operation invocations.*

Var QE: List of N^e;
Var QOp: List of $\langle Op, Rs, NRs\rangle$;
Var $CE : N^e$; (* *The current entity node to be processed.* *)
Var $COp : N^o$; (* *The current operation node to be processed.* *)
Var $EChd$: List of N^e;
Var $OpChd$: List of N^o;
Step 1. (* *Initialisation* *)
$QOp := \emptyset$;
$QE := QE + \{x \in N^e | InDegree(x) = 0\}$;
if Test case t has at least one condition **then**
    $QE := QE + \{x | x$ is a start entity node of a condition$\}$;
else
    $QE := QE + \{x | x$ is a start entity node of the equations$\}$.
end if
Step 2.
  CE := $Get(QE)$;
while $CE \neq$ nil **do**
    if $OutDegree(CE) = 0$ **then**
        $GenCode($"*Output*" $+ Object(CE))$;
    else if $OutDegree(CE) = 1$ **then**
        $OpChd := NextOps(CE)$;
    else if $OutDegree(CE) > 1$ **then**
        $EChd := NextEs(CE)$;
        $OpChd := NextOps(CE)$;
        Var $s := Sort(CE)$;
        Var Copies: List of $N^e := \{nd \in EChd | Sort(nd) = s\}$;
        if $||Copies|| > 1$ **then**
            if s is not controllable **then**
                Terminate (* *The TEG is not monolithic.* *)
            else(* ***Assert:*** s *is controllable* *)
                (* *Generate code of saving CE* *)
                Var String x := NewVar(s);
                $GneCode("Var" + x + " : " + s + ";")$;
                $GenCode(x + " := " + Object(CE))$;
            end if
        end if
    end if
    Order nodes in $OpChd$ such that attribute nodes are listed first;
    for Each $COp \in OpChd$ **do**
        (* *Process Operation Node* COp *);
        if $InDegree(COp) = 1$ **then**
            if $OpOutEs(COp) \in Copies$ **then**
                $GenCode(Label(CE) + " := " + x)$;
            end if
            $GenCode(Label(COp) + "(" + Object(CE) + ")")$;
            $EChd := OpOutEs(COp)$;
            Add all elements of $EChd$ to the front of QE;
        else(* ***Assert:*** $InDegree(Op) > 1$ *)
            Search COp in the QOp;
            if $\langle COp, Rs, NRs\rangle \in QOp$ **then**
                $Rs := Rs + CE$;
                $NRs := NRs - CE$;
                if $NRs = \emptyset$ **then**
                    if $OpOutEs(COp) \in Copies$ **then**
                        $GenCode(Label(CE) + " := " + x)$;
                    end if
                    $GenCode(COp + "(" + Rs + ")")$;
                    Remove COp from QOp;
                    $EChd := OpOutEs(COp)$;
                    Add all elements of $EChd$ to the front of QE;
                end if
            else(*Assert: $\langle COp, Rs, NRs\rangle \notin QOp$*)
                Var $Rs := \langle CE\rangle$;
                Var $NRs := OpInEs(COp) - CE$;
                Add $\langle COp, Rs, NRs\rangle$ into QOp;
            end if
        end if
    end for
    CE := $Get(QE)$;
end while