A Note on Test Oracles and Semantics of Algebraic Specifications

Hong Zhu
Department of Computing, School of Technology, Oxford Brookes University
Wheatley Campus, Oxford OX33 1HX, UK, Email: hzhu@brookes.ac.uk

Abstract

Algebraic testing is an automated software testing method based on algebraic formal specifications. It tests if a program correctly implements an algebraic specification by checking if the equations of the specification are satisfied. One of the key techniques of algebraic testing is the use of observation contexts as a test oracle so that the comparison of values of structured data can be realised by comparing values of simple data types. This leads to a behavioural semantics of algebraic specifications and divides sorts into observable and non-observable. To represent the structure of object-oriented programs and to facilitate incremental integration testing, this paper extends the notion of observable sorts by introducing a partial ordering between sorts to represent the importation relation between classes. In this framework, the validity of test oracles is formally proved in final algebra semantics.

1. Introduction

Algebraic specification emerged in the 1970s as a formal method for specifying abstract data types in an implementation-independent style; see, e.g. [1, 2]. In the past thirty years, it has developed into an important formal software development method [3, 4, 5]. The uses of algebraic specification in software testing can be back dated to early 1980s [6]. Since then, significant progresses have been made in the techniques that support automated software testing using algebraic specification through a number of researchers' work. However, it has not been thoroughly investigated in the literature on how algebraic specifications should be written and understood as the input to automated testing tools so that the power of algebraic testing can be fully realised. In this paper, we approach this problem from both the syntactic structures and the formal semantics of algebraic specifications.

2. Related works and open problem

In early 1980s, Gonnon, McMullin and Hamlet developed a compiler based system called DAISTS to use algebraic specifications in testing abstract data types implemented in procedural programming languages [6]. Their basic idea was based on the observation that each term of a given signature has two interpretations in the context of software testing. First, a term represents a sequence of calls to the operations specified in the algebraic specification. These operations were usually implemented by procedures and functions in procedural languages. When the variables in a term were replaced by constants, the sequence of calls represents a test execution of the program, where the constants substituted for the variables constitute the test case. Second, a term also represents a value, i.e. the result of the execution. Therefore, checking whether an equation is satisfied by an implementation on a test case meant to execute the operation sequences of the terms on both sides of the equation and then to compare the results. If the results are equivalent, the program is correct on this test case; otherwise, the implementation had errors. The input to DAISTS was written in a specifically designed language called SIMPL-D. An input to DAISTS consists of a set of algebraic axioms of an abstract data type and a set of test cases, which were ground terms to be substituted into the axioms to replace the variables in the axioms. The axioms were used as test drivers that invoke the implemented functions of the operators. Each axiom was tested on manually scripted test cases and the results from both sides of the axiom were compared by calling a TypeEquality function according to the data type of the result. For basic data types such as integer, DAISTS system used the predefined equality functions as the TypeEquality function. For user defined abstract data types, the TypeEquality function was manually programmed.

In late 1980s, Gaudel *et al.* developed a theory and a method of specification-based software testing [7, 8]. Among the most important contributions of Gaudel and her colleague's work was the use of observation contexts, which enables automatic comparisons of structured values without manually programming equality functions. The basic idea of observation contexts was that, to compare the equality of two structured values, a number of operations could be applied on the structured values to generate simpler values that can be compared effectively, such as values in predefined data types. Such operations are called the observation contexts in the behavioural theories of algebraic specifications; see e.g. [3]. In a theory of behavioural algebraic specifications, sorts in an algebraic specification can be divided into two disjoint types: observable sorts and hidden sorts. Observable sorts represent those basic data

types on which the equivalence between values can be effectively determined. Hidden sorts, or unobservable sorts, represent structured data types or states of objects whose values cannot be determined of equivalence effectively. Another major contribution was the concept of test contexts, which consists of a hypothesis about the software under test and a set of test cases selected using a certain test criterion. It enabled theoretical analysis of how a test case selection criterion is related to the correctness of the software. A test hypothesis explicitly expresses the condition under which a program is correct if it passes the test on all the test cases selected according to a certain criterion. In particular, they defined exhaustive test set as all ground instances of the axioms and claimed that a program is correct if it pass the test on exhaustive test set. The most appealing feature of Gaudel's approach is that formal specifications can be used to automatically generate test cases as well as test oracles to determine if the program produces correct output. It can achieve a high degree of test automation.

Algebraic testing received much attention since 1990s in the context of class testing of object-oriented programs. In early 1990s, Frankl and Doong studied the effectiveness of testing object-oriented programs based on algebraic specifications [9, 10]. They adopted a notation of representation of algebraic specifications that are suitable for object-oriented implementation and developed an algebraic specification language called LOBAS and a tool called ASTOOT. They conducted case studies to assess the practical usability of Gaudel's method and technique. They pointed out that the exhaustive test set of Gaudel's method is inadequate to rule out undesirable implementations. One of their most important contributions to the method is the extension of test cases to include negative test cases, which consists of two terms that are supposed to generate non-equivalent results. There are two implicit fundamental assumptions underlying Frankl and Doong's algorithm of test case generation, as pointed out by Chen, Tse and Chen [11] recently. Given an algebraic specification and a complete implementation of the specification, let u be any ground term and $\Theta(u)$ denote the object produced by the method sequence corresponding to u.

Assumption 1. If u and v are equivalent in the specification, but the objects $\Theta(u)$ and $\Theta(v)$ are observably non-equivalent, then the implementation is incorrect.

Assumption 2. If u and v are not equivalent in the specification, but $\Theta(u)$ and $\Theta(v)$ are observably equivalent objects, then the implementation is incorrect.

Frankl and Doong defined the equivalence relation between two ground terms as the transformability from one term into another by using the axioms as rewriting rules, provided that the specification is canonical.

These assumptions are the bases of error detection for algebraic testing. However, Chen, Tse and Chen pointed out that the assumptions are not always true and Frankl and Doong's algorithm for generating negative test cases was incorrect [11]. They further developed the theory and method of automatic derivation of test oracles based on observation contexts [11, 12]. They introduced the concept of observable equivalence relation between terms in a specification. They suggested the replacement of the above assumptions with the following two criteria to define what a correct implement meant.

Equivalence Criterion: For all ground terms u and v, u and v are observationally equivalent in the specification implies that objects $\Theta(u)$ and $\Theta(v)$ are observationally equivalent.

Non-equivalence Criterion: For all ground terms u and v, u and v are observationally non-equivalent in the specification implies that objects $\Theta(u)$ and $\Theta(v)$ are observationally non-equivalent.

Chen, Tse and Chen proved that the equivalence criterion is equivalent to the assumption 1, and observational equivalence is weaker than normal equivalence in general. They also gave a counterexample to show that the non-equivalence criterion is not equivalent to assumption 2. A question remains open that under what condition observational equivalence is the same as the normal equivalence relation.

The use of the above criteria to replace Frankl and Doong's assumptions is actually moving from classical initial semantics of algebraic specifications to behavioural semantics. Chen, Tse and Chen's work indicates that how to understand the semantics of algebraic specifications is vital to software testing, especially to the validity of test oracles. More precisely, whether a test oracle is valid depends on the semantics of the algebraic specification. The question that what is a correct implementation of a formal specification has been a central problem in the research on formal specifications, especially in the research on algebraic specifications; see e.g. [13] for a survey. Guadel and her colleagues explicitly referred to Hennicker's work [14] on the observational implementation of algebraic specifications [7, 8,15].

Generally speaking, a behavioural specification consists of a pair (*SP*, *Obs*), where *SP* is a formal specification and *Obs* describes what is observable in *SP*. There are a number of different approaches to the description of *Obs*. The first approach, and perhaps the most common approach, is to distinguish a subset of the sorts in *SP* as observable and others are considered as not observable (or hidden sort), e.g. [16,3]. This is the approach that Guadel and her colleagues have used as well as other existing work on algebraic testing. However, distinction between observable sorts from non-observable sorts was considered as inadequate. Hence, Hennicker used a predicate $Obs_S(x)$ on each sort S to describe a subset of the values of the sort S are observable [14]. In [17], *Obs* was defined as a subset of terms so that terms

in *Obs* is considered as observable computations. To study behaviour specification in an institute independent framework, Sannella and Tarlecki used another approach and defined *Obs* as a set of formulae [18]. This is the most general form of behavioural specification. Similar to classical theories of algebraic specifications, the notion of implementation of a behavioural specification can be considered as a model (or a set of models) with certain specific properties that satisfies the axioms of the algebraic specification. That is, algebra A behaviourally satisfies an axiom with respect to a given set of observations iff it satisfies all its observable consequences. Given an equation e: $\forall X.\ t_1=t_2$, the observable consequences of e in the algebra A are all the equations $c[\sigma(t_1)] = c[\sigma(t_2)]$ for all contexts c and substitutions $\sigma: X \rightarrow T_\Sigma$, such that $c[\sigma(t_1)]$ and $c[\sigma(t_2)]$ denote observable values in A. As pointed out in [13], there are subtle differences between the different approaches to the definitions of what are observable and some may lead to difficult semantic problems. In the past a few years, researchers in the area of algebraic specifications focus on the problems about how to define the semantics of behavioural specifications and the logic for the uses of such specifications in reasoning about software properties; see e.g. [19, 20].

A common weakness of algebraic testing techniques is that software is tested in a 'big bang' approach, i.e. all classes of a system is tested all together without employing any incremental integration strategy. This seriously limited the practical usability of the testing method. This paper addresses this problem by proposing an approach to the organisation of algebraic specifications to match the structures of object-oriented software systems. Equations in an algebraic specification are divided into groups that each group represents a class in object-oriented system. A partial ordering between sorts is introduced to represent the importation relationship between classes. This partial ordering generalises the notion of observable sorts and supports incremental integration testing.

The remainder of the paper is organised as follows. Section 3 defines algebraic specification with partial ordering between sorts. Section 4 proves that the validity of observation contexts in such structured algebraic specification within final algebra semantics. Section 5 is the conclusion of the paper and discusses future works.

3. Structuring algebraic specifications

An object-oriented program usually consists of a number of classes. A class cannot be executed in isolation without 'importing' its supporting classes, which may be a types of the class' attributes or parameters of the methods, the result value of a method, or a class of a local variable used to implement a method. Obviously, such importing/supporting relationship is a pre-order. At the lowest level, there are a number of pre-defined classes, such as those of basic data types like Boolean, Integer and Real. The most important property of such importing/supporting relationship is that the importing class does not modify the semantics of the supporting classes. This is the property that distinguishes the relationship from inheritance. In software testing practices, the import relationship forms the bases of defining integration strategies, which include top-down and bottom-up strategies. It enables the reduction of the complexity of integration of complicated software systems through incremental integration. Notice that, partial ordering between sorts has been investigated to represent the concepts of inheritance and subtypes. Such partial ordering is significantly different from what is introduced in this paper.

By considering sorts as data types that are implemented by classes, one would expect that an algebraic specification is decomposed into units of similar relationship. Indeed, modern algebraic specification languages such as CafeOBJ and OBJ3 provide modular structure to group equations into modules. In addition to classic enrichment or extension operations on modules [4, 5], these languages also provide a protected importation operation on modules. A module of algebraic specification enriches or extends other module(s) to compose modules together and to build a new module on the base of existing modules. The semantics of such an extension is to put all the sorts, operations and their axioms together. One module that extends another may have additional operations and/or axioms defined for the sorts that are already defined in existing modules. Therefore, enrichment may change the semantics of existing modules. It is more like the inheritance relationship between classes. In contrast, protected importation operation protects the semantics of imported module unchanged in the new module. This resembles the importation relationship between classes. To use this protected importation facility more effectively in object-oriented software development, we further suggest that each module in the specification should consist of one main sort that represents the values of the objects of the class and a number of supporting sorts protectively imported from other modules. The axioms of the module should not modify the semantics of the supporting sorts that are defined in protectively imported modules. In testing a class, only the axioms for the main sort need test rather than the axioms of supporting sorts. At the lowest level of this protective importation hierarchy are basic modules that are directly implemented by basic classes of programming language. We assume that a basic class is correctly implemented by the system, and correctly selected for the specification module whose main sort corresponds to the class. Such basic classes must be testable, i.e. observable, in the following sense.

Definition 3.1 (Observable sort)

In an algebraic specification $<\Sigma, E>$, a sort s is called an *observable sort*, if there is an operation $\_ == \_ : s \times s \rightarrow Bool$ such that for all ground terms τ and τ' of sort s,

$$E \vdash (\ (\tau == \tau') = true) \quad \Leftrightarrow \quad E \vdash (\ \tau = \tau'\ )$$

where Σ is a signature, which consists of a set S of sorts and a finite family $<\Sigma_{w,s}>$ of disjoint finite sets indexed by $S^* \times S$. $\Sigma_{w,s}$ is the set of operator symbols of type $<w, s>$. An algebra A is a correct implementation of an observable sort s, if for all ground terms τ and τ' of sort s,

$$A \models (\tau = \tau') \quad \Leftrightarrow \quad A \models ((\tau == \tau') = true) \quad \square$$

The distinction between main sorts from supporting ones does not only decide which axioms are to be checked, but also plays a significant role in the derivation of test oracles. For the sake of space, subsequently, we use 'importation' for 'protected importation'.

Definition 3.2 (Importation relation on sorts)

The import relation is a pre-order $\prec$ on the set S of sorts that satisfies the following conditions

(1) For all sorts $s \in \Sigma$, s is an observable sort, if there is no sort $s' \prec s$;

(2) For all sorts $s, s' \in \Sigma$, $s' \prec s$ and s is an observable sort imply that s' is also an observable sort.

We say that sort s_1 is a sort that supports sort s_2, or s_2 imports s_1, if $s_1 \prec s_2$. We also say that s_1 directly supports s_2 if $\neg \exists s'' \in \Sigma . (s' \prec s'' \wedge s'' \prec s)$. □

Having defined the notion of supporting sorts, classification of operators in a canonical algebraic specification can be formally defined as follows.

Definition 3.3 (Creator, constructor, transformer, and observer)

An operator $\sigma : w_1 \times \ldots \times w_n \rightarrow c$ is called a *creator* of sort c, if for all i=1, 2, …, n, $w_i \neq c$ and $w_i \prec c$. In particular, when n=0, $\sigma : \rightarrow c$ is a constant creator of sort c.

An operator $\sigma : w_1 \times \ldots \times w_n \rightarrow c$ is called a *constructor* of sort c, if there is at least one $i \in \{1, 2, \ldots, n\}$, such that $w_i = c$, and for all i=1, 2, …, n, $w_i = c$ or $w_i \prec c$, and the operator σ can appear in at least one normal form of ground terms.

An operator $\sigma : w_1 \times \ldots \times w_n \rightarrow c$ is called a *transformer* of sort c, if there is at least one $i \in \{1, 2, \ldots, n\}$, such that $w_i = c$, and for all i=1, 2, …, n, $w_i = c$ or $w_i \prec c$, and the operator σ cannot appear in any normal form of ground terms.

An operator $\sigma : w_1 \times \ldots \times w_n \rightarrow s$ is called a *observer* of sort c, if for all i=1, 2, …, n, $w_i = c$ or $w_i \prec c$, there is at least one $i \in \{1, 2, \ldots, n\}$, such that $w_i = c$, $s \neq c$ and $s \prec c$. □

Here, we also use the LOBAS notation, i.e. $w_1 = c$ for an operator to be a constructor, transformer, or observer to indicate that the first operand is the state of the object. We will assume that the specifications are canonical in the sequel.

An operator in a canonical algebraic specification is either a creator, or a constructor, or a transformer, or an observer. It can only be one of these types. The axioms of an algebraic specification should also preserve the pre-order of 'support' relation by satisfying the following conditions.

(3) For all s and $s' \in \Sigma$, $s' \prec s$ and s' directly supports s, there is an observer σ of sort s such that $\sigma : w_1 \times \ldots \times w_n \rightarrow s'$;

(4) For all conditional equations ($\tau = \tau'$, if $\tau_1 = \tau_1' \wedge \ldots \wedge \tau_k = \tau_k'$), for all i = 1,..., k, $s_i \prec s$ or s_i is observable, where s_i is the sort of τ_i and τ_i', s is the sort of τ and τ'.

Definition 3.4 (Well structured specification)

A canonical specification $<\Sigma, E>$ is well structured with respect to an importation relation $\prec$ on the sorts, if it satisfies properties (1) ~ (4). □

Example 1. Consider the following algebraic specification of natural number queues.

```
Spec QUEUE;
    Protected Import Nat from NAT, Bool from BOOL;
    Sort: Queue;
    Operators:
        Create:    –> Queue;
        _.Put(_): Queue x Nat –> Queue;
        _.Front:  Queue     –> Nat;
        _.Get:    Queue     –> Queue;
        _.Is-Empty:  Queue –> Bool;
        _.Length:    Queue –> Nat;
    Axioms:  Var Q: Queue, N, M: Nat;
        Create.Length = 0;
        Q.Put(N).Length = Q.Length+1;
        Q.Put(N).Front = N
        Create.Is-Empty = True;
        Q.Put(N).Is-Empty = False;
        Create.Put(N).Get = Create;
        Q.Put(N).Put(M).Get = Q.Put(N).Get.Put(M);
End QUEUE;
Spec NAT;
    Protected Import Bool from BOOL;
    Sort: Nat;
    Operators:
        0:      –> Nat;
        _+1:   Nat –> Nat;
        _==_: Nat x Nat  –> Bool;
        _>_:   Nat –> Bool;
    Axioms: … (* Details are omitted *)
End NAT;
Spec BOOL;
    Sort: Bool;
    Operators:
        True :     –> Bool;
        False:     –> Bool;
        _==_:    Bool x Bool –> Bool;
        … (* Details are omitted for the sake of space *)
End BOOL.
```

The specification contains three modules. Each module only defines one sort, which can be implemented by one class. The module QUEUE contains equations that define

the operations on natural number queues. It imports sort Nat from module NAT and Bool from module BOOL, which contain the axioms of natural numbers and Boolean values, respectively. NAT also imports sort Bool from module BOOL for its axioms. Therefore, a pre-order on sorts can be defined as follows: Bool $\prec$ Nat $\prec$ Queue. Assume that the sort Nat is implemented by a pre-defined class *Cardinal* and there is an operation for test equivalence between two cardinal values. Nat is, then, an observable sort. Similarly, we assume Bool is an observable sort. length(x), is_empty(x) and front(x) are observers of Queue. With proper equations, we can see that Put is a constructor and Get is a transformer of the sort Queue. Create is a creator of queue. Since there are only three sorts in the specification of natural number queues and sort Nat and Bool support Queue, it is easy to see that the specification of queues is well structured. □

Let $<\Sigma,E>$ be an algebraic specification, and a Σ-algebra A be an implementation of the specification. We assume that the specification is well structured and $\prec$ is the importation relation between the sorts. The following definition adapts the concept of observable context sequences introduced in [11].

Definition 3.5 (Observable context)

A context $C[\ldots]$ of a sort c is a term C with one occurrence of a special variable □ of sort c. The value of a term t of sort c in the context of $C[\ldots]$, written as $C[t]$, is the term obtained by substitute t in to the special variable □. An *observable context oc* of sort c is a context of sort c and the sort of the term $oc[\ldots]$ is $s \prec c$. To be consistent with our notation for operators, we write $\_.oc: c \rightarrow s$ to denote such an observable context $oc[\ ]$.

An *observable context sequence* of a sort c is the sequential composition $\_.oc_1.oc_2. \ldots.oc_n$ of a sequence of observable contexts $oc_1, oc_2, \ldots, oc_n$, where $\_.oc_1: c \rightarrow s_1$, $\_.oc_i: s_{i-1} \rightarrow s_i$, for all $i = 2,\ldots,n$. An observable context sequence is *primitive*, if the s_n is an observable sort. □

In other words, an observable context *oc* of sort c is either an observer of the sort c, or a context whose top-most operator is an observer of the sort c. The general form of an observable context *oc* is as follows:

$$\_.f_1(\ldots).f_2(\ldots).\ldots.f_k(\ldots).obs(\ldots)$$

where $f_1, \ldots, f_k$ are constructors or transformers of sort s_c and *obs* is an observer of sort c, $f_1(\ldots), \ldots, f_k(\ldots)$ are ground terms. A primitive observable context produces a value in an observable sort.

Example 2. For the operators on queues given in Example 1, the following terms are observable contexts of sort Queue: length(□), front(□), is_empty(□), front(get^k(□)), length(get^k(□)) and is_empty(get^k(□)), for all $k = 1, 2, \ldots,$ □

It is worth noting that there are usually an infinite number of different observation contexts for a given algebraic specification. Obviously, for a well structured system, we have the following property.

Lemma 3.1 In a well structured system, we have that:
(1) For any sort c, all observable context sequences of sort c are of finite length.
(2) For all observable context sequences *ocs*, *ocs* can be extended to a primitive observable context sequence.

Proof. It follows the facts that the set of sorts is finite and the support relation is a pre-order on the sorts. □

Existing algebraic testing methods use primitive observation contexts for testing all equations [7~12]. This may cause repeated checking the equivalences of a large number of values of observable sorts. The importation relationship can be used to o avoid such situations by deploying a top-down strategy of test case generation. The following is an algorithm that illustrates how a top-down strategy can be implemented to generate positive test cases.

Algorithm (Generation of positive test cases)

INPUT:
- *Spec*: a well structured algebraic specification, with a set S of n sorts and an importation relation $\prec$ on S.
- CC: $S \times Spec \rightarrow Context$: a criterion for the generation of observable contexts. For all sorts $s \in S$, $CC(s, Spec)$ is a finite set of observable contexts of sort s.
- TC: $S \times Spec \rightarrow P(Term \times Term)$: a test criterion for the generation of test cases. For all sorts $s \in S$, $TC(s, Spec)$ is a finite set of test cases. Each test case consists of two ground terms of sort s.

VARIABLES:
- T_i, i=1, 2, …, n : To store the set of test cases of sort s_i;
- C_i, i=1, 2, …, n : To store the set of observable contexts of sort s_i.

OUTPUT:
- T: A set of observable test cases generated from *Spec* that satisfy TC and CC.

BEGIN

Step 1. (* Topologic sorting of sorts into descending order*)
Generate a sequence $<s_1, s_2, \ldots, s_n>$ that contains all sorts in S and for all $s_i, s_j \in S$, $s_i \prec s_j$ implies that $j<i$;

Step 2. (* Initialisation *)
$T := \varnothing$;
FOR i:=1 TO n DO $T_i := \varnothing$;

Step 3. (* Generate observable contexts C_i for sort s_i using context generation criterion CC *)
FOR i:=1 TO n DO $C_i := CC(s_i, Spec)$;

Step 4. (* Generate observable test cases for all sorts *)
FOR i:=1 TO n DO
BEGIN (* Apply test case generation criterion TC to *Spec* for sort s_i *)
$T_i := T_i \cup TC(s_i, Spec)$;
IF sort s_i is not observable
THEN (* Apply observable contexts *)

FOR all $C[\ldots] \in C_i$ and all $(u, v) \in T_i$ DO
$T_j := T_j \cup \{(C[u], C[v])\}$,
where $sort(C[\ldots]) = s_j$
END;
END;
Step 5. (* Collect observable test cases *)
FOR i:=1 TO n DO
IF s_i is observable THEN T:=$T \cup T_i$;
Step 6. Output(T);
END. □

In the above algorithm, duplicated test cases are automatically deleted because of set union operation.

4. Validation of test oracles in final algebra

Chen, Tse and Chen [11] noticed that it is possible that the equivalence of two ground terms u_1 and u_2 cannot be proved in equational logic from the axioms while the difference between the two terms cannot be detected by observation contexts. They gave an example of such situation. This raised the validity question of the observable context test oracle. This section proves that test oracles based on observable contexts are valid in final algebra semantics.

4.1 Observable equivalence

Definition 4.1 (Observational equivalence of terms) [11]

Given a canonical specification <Σ, *E*>, two ground Σ-terms u_1 and u_2 are said to be *observational equivalent* (denoted by '$u_1 \sim_{obs} u_2$') if and only if the following condition is satisfied.

(1) The normal forms of u_1 and u_2 are identical, if the sort s of u_1 and u_2 is observable; otherwise,

(2) for all observation contexts *oc* of sort s, $u_1.oc$ and $u_2.oc$ are observationally equivalent. □

The following two lemmas are from Chen, Tse, *et al*. Their proofs can be found in [11].

Lemma 4.1 Given a canonical specification <Σ, E>.

(1) Two ground Σ-terms u_1 and u_2 of an observable sort s are observationally equivalent, if and only if their normal forms are identical.

(2) Two ground Σ-terms u_1 and u_2 of a non-observable sort s are observationally equivalent, if and only if for all primitive observable context sequence *ocs*, the normal forms of $u_1.ocs$ and $u_2.ocs$ are identical. □

Lemma 4.2 (Subsume relationship theorem)

Given a canonical specification <Σ, *E*>, for all ground terms τ and τ' of same sort, we have that $E|-\tau=\tau'$ implies that $\tau \sim_{obs} \tau'$. □

Notice that, the converse of Lemma 4.2 is not always true. The example of bank account specification given in [11] is a counter-example of the converse of Lemma 4.2.

4.2 Characteristic theorem

Chen, Tse and Chen's Subsume Relation Theorem (Lemma 4.2) states that observational equivalence is not always the same as the equivalence relation in the initial algebra. We now prove that observational equivalence is the same as the normal equivalence relation in the final algebra.

Lemma 4.3 The relation $\sim_{obs}$ is an equivalence relation on the set W_Σ of ground Σ-terms.

Proof. The statement follows Lemma 4.1. The proof is straightforward. □

Theorem 4.1 (Congruence theorem of observationally equivalence)

For a well structured canonical specification <Σ, *E*>, the observational equivalence relation $\sim_{obs}$ is congruent with the operations in the specification <Σ, E>.

Proof. We only need to prove that for all context $C[\ ]$, $u_1 \sim_{obs} u_2$ implies that $C[u_1] \sim_{obs} C[u_2]$.

If the context $C[\ ]$ itself is a primitive observable context sequence, then by Definition 4.1, we have that $C[u_1]$ and $C[u_2]$ have identical normal form. Being a primitive observable context sequence, the sort of $C[...]$ is observable. By Definition 4.1, we have that $C[u_1] \sim_{obs} C[u_2]$.

If the context $C[\ ]$ is not a primitive observable context sequence, by the definition of well structured systems, the context can be extended to primitive observable context sequences ocs. For all such primitive sequences ocs, $C[u_1]$.ocs can be written in the form of u_1.C.ocs. By Lemma 4.1, since $u_1 \sim_{obs} u_2$, the normal form of u_1.C.ocs is identical to the normal form of u_2.C.ocs. By Lemma 4.1, we have that u_1.C $\sim_{obs} u_2$.C. That is $C[u_1] \sim_{obs} C[u_2]$. □

From the proof of Theorem 4.1, it is easy to see the attribute equivalent relation $\sim_{att}$ defined in [11] is not congruent to the operations in specification <Σ, *E*>, because the context $C[...]$ can be a constructor rather than an observer.

Definition 4.2 (E-congruence)

A congruence ~ on algebra A is said to be an E-congruence, if for each conditional equation in E,

$$\tau = \tau' \text{, if } (\tau_1 = \tau_1') \wedge \ldots \wedge (\tau_k = \tau_k')$$

and for all assignments φ in the algebra A, we have that $[\![\tau]\!]_\varphi \sim [\![\tau']\!]_\varphi$, if $[\![\tau_1]\!]_\varphi \sim [\![\tau_1']\!]_\varphi \wedge \ldots \wedge [\![\tau_k]\!]_\varphi \sim [\![\tau_k']\!]_\varphi$.

Theorem 4.2 (E-congruence theorem)

Given any well structured specification <Σ, *E*>, the observational equivalence relation $\sim_{obs}$ defined on ground terms is *E*-congruence.

Proof. We prove by structured induction on the sort s of the terms τ and τ' in the equation

$$\tau = \tau' \text{, if } (\tau_1 = \tau_1') \wedge \ldots \wedge (\tau_k = \tau_k')$$

Let s_i be the sort of the terms τ_i and τ'_i in the above equation. Let μ be any ground substitution.

(1) If the sort s is observable, by Definition 3.4, for all $i = 1, 2, \ldots, k$, s_i is observable. By Lemma 4.1, $\mu(\tau_i) \sim_{obs} \mu(\tau'_i) \Leftrightarrow E|- \mu(\tau_i) = \mu(\tau'_i)$. Thus, $E|- \mu(\tau) = \mu(\tau')$. Since s is observable, we have that $\mu(\tau) \sim_{obs} \mu(\tau')$.

(2) Suppose that for all sorts s' that $s' \prec s$ or s' is observable, we have that for all terms τ_1 and τ_2 of sort s', $\mu(\tau_1) \sim_{obs} \mu(\tau_2) \Rightarrow E|- \mu(\tau_1) = \mu(\tau_2)$. Then, we have that

$$\mu(\tau_1) \sim_{obs} \mu(\tau'_1) \wedge \ldots \wedge \mu(\tau_k) \sim_{obs} \mu(\tau'_k) \Rightarrow$$
$$E \mid -\mu(\tau_1) = \mu(\tau'_1) \wedge \ldots \wedge E \mid -\mu(\tau_k) = \mu(\tau'_k).$$

Therefore, we have that $E|- \mu(\tau) = \mu(\tau')$ in equational logic. Hence, by Lemma 4.2, $\mu(\tau) \sim_{obs} \mu(\tau')$. □

Corollary of Theorem 4.2**.**

Given a well structured canonical specification <Σ, *E*> and its final algebra *B*, for all ground terms τ and τ', $\tau \sim_{\text{obs}} \tau'$ imply that $B|=\tau=\tau'$.

Proof. Let *B* be the final algebra of all <Σ, *E*>-algebras. By the property of final algebra [21], we have that, for all *E*-congruence relation ~ on W_Σ which is not a unit algebra, $\tau \sim \tau'$ imply that $B|=\tau=\tau'$. The statement immediately follows the fact that $\sim_{\text{obs}}$ is an *E*-congruence and not unit as proved in Theorem 4.2. □

Theorem 4.3 (Characteristic theorem)

The term algebra $W_\Sigma/\sim_{\text{obs}}$ is the final *E*-algebra.

Proof. Let <Σ, *E*> be a well structured canonical specification. Let *B* be the final algebra of <Σ, *E*>.

By the corollary of Theorem 4.2, $\tau \sim_{\text{obs}} \tau'$ implies $B|=\tau=\tau'$. The following proves that for all ground terms τ and τ', $B|=\tau=\tau'$ implies that $\tau \sim_{\text{obs}} \tau'$. Let $\tau, \tau' \in W_{\Sigma,s}$ and $B|=\tau=\tau'$.

(1) If the sort s is observable, by Definition 3.1, we have that $B|=\tau=\tau'$ if and only if $E|-\tau=\tau'$. Since the specification is canonical, the normal forms of τ and τ' are identical.

(2) If the sort s is not observable, the statement $B|=\tau=\tau'$ is equivalent to the statement that $[\![\tau]\!]_B = [\![\tau']\!]_B$. Let $\_.w$ be any primitive observable context sequence. We have that $[\![\tau]\!]_B.[\![w]\!]_B = [\![\tau']\!]_B.[\![w]\!]_B$. Thus, $[\![\tau.w]\!]_B = [\![\tau'.w]\!]_B$, or equivalently, $B|= \tau.w=\tau'.w$. Since the sort of the terms $\tau.w$ and $\tau'.w$ are observable, by (1) above, we have that $\tau.w$ and $\tau'.w$ have identical normal forms.

By Lemma 4.1, in both cases, we have that $\tau \sim_{\text{obs}} \tau'$. □

4.3 Testing final algebras

To understand how observational equivalence can be applied to testing final algebras, we need to know if two observationally equivalent terms will be observationally equivalent objects.

Definition 4.3 (Observably equivalent objects)

Two objects a_1 and a_2 of sort s are *observably equivalent*, written $a_1 \approx_{\text{obs}} a_2$, if they satisfy the following conditions.

(1) $a_1 == a_2$, if s is an observable sort;

(2) for all observable contexts *oc* of the sort s, $a_1.oc \approx_{\text{obs}} a_2.oc$, if s is not an observable sort. □

Let τ and τ' be any given ground terms. The validity requirements require that, first, $[\![\tau.oc]\!]_A \approx_{obs} [\![\tau'.oc]\!]_A$, for all observable context *oc*, if the semantics of the algebraic specification <Σ, *E*> requires that a correct implementation $A|=\tau=\tau'$. Second, for some observable context oc, $[\![\tau.oc]\!]_A \not\approx_{obs} [\![\tau'.oc]\!]_A$, if the semantics of the algebraic specification <Σ, *E*> requires that a correct implementation $A|= \tau\neq\tau'$. The following theorem formally proves these properties for final algebra semantics.

Theorem 4.4 (Validity theorem)

Let <Σ, *E*> be a well structured canonical specification. An algebra *A* of the specification is the final algebra, implies that *A* satisfies the following conditions.

(1) *Equivalence criterion*: For all ground terms τ and τ', $\tau \sim_{\text{obs}} \tau'$ implies that $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$;

(2) *Non-equivalence criterion*: For all ground terms τ and τ', not ($\tau \sim_{\text{obs}} \tau'$) implies that $[\![\tau]\!]_A \not\approx_{obs,A} [\![\tau']\!]_A$;

Proof. We only need to prove that *A* is the final algebra implies that for all ground terms τ and τ', $\tau \sim_{\text{obs}} \tau' \Leftrightarrow [\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$.

Note that, first, *A* is isomorphic to $W_\Sigma/\sim_{\text{obs}}$. Let θ be the isomorphism between them. Second, for all ground terms τ, $[\![\tau]\!]_A = \theta([\tau]_\sim)$, where $[\tau]_\sim$ is the equivalence class of τ under the relation $\sim_{\text{obs}}$. For all ground terms τ and τ', we have that $\tau \sim_{\text{obs}} \tau' \Leftrightarrow [\tau]_\sim = [\tau']_\sim \Leftrightarrow \theta([\tau]_\sim) = \theta([\tau']_\sim) \Leftrightarrow [\![\tau]\!]_A = [\![\tau']\!]_A \Leftrightarrow A|= \tau=\tau' \Rightarrow [\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$. Thus, $\tau \sim_{\text{obs}} \tau' \Rightarrow [\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$.

To prove that $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!] \Rightarrow \tau \sim_{\text{obs}} \tau'$, consider the sort s of the terms τ and τ'. If s is observable, by Definition 3.1 and Definition 4.3, we have that $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!] \Rightarrow A|= \tau=\tau'$. By Definition 3.1 and Definition 4.1, we have that $\tau \sim_{\text{obs}} \tau'$. If the sort s is not observable, by Definition 4.3, $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]$ implies that that for all primitive observable context sequences *w*, $A|= \tau.w = \tau'.w$. Since the sort of the terms $\tau.w$ and $\tau'.w$ is observable, we have that $E|- \tau.w = \tau'.w$. By Definition 4.1, we have that $\tau \sim_{\text{obs}} \tau'$. □

This theorem formally proves that the observation context oracle satisfies the validity requirements A for the final algebra semantics. It states that to test a final algebra implementation *A* against a well structured canonical specification <Σ, *E*>, we need to check for all ground terms τ and τ' of the same sort so that we can conclude that *A* is a correct implementation, if the following conditions are true.

(1) if $\tau \sim_{\text{obs}} \tau'$ is required by the specification, the test oracle

reports that $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$;

(2) if $\tau \neq_{obs} \tau'$ is required by the specification, the test oracle reports that $[\![\tau]\!]_A \not\approx_{obs,A} [\![\tau']\!]_A$.

How to check these conditions has been discussed in [9~11]. This paper proves that the conclusions one can draw are only valid in final algebra semantics.

5. Conclusion

In this paper, we extended the notion of observable sorts by introducing a pre-order relation between sorts and organising algebraic specifications by dividing equations into groups to match the importation relationship between classes. We proved that in this framework test oracles based on observation contexts satisfy the validity requirements for correct implementations of well-structured canonical algebraic specifications if and only if the semantics of the specification is the final algebra. Otherwise, the validity requirement is not necessarily satisfied.

For future research, we believe that the theories of observational algebraic specifications can be applied to a wider range of software systems such as concurrent systems, because concurrency and non-determinism can be treated in behavioural theories naturally [3]. We are also investigating the relationship between the theories of behavioural algebraic specifications and a more general theory of behavioural observation in software testing developed independently in our previous work [22, 23, 24]. Our general theory of software testing considers each testing methods consists of two components: a test adequacy criterion and a behaviour observation scheme. Mathematical structures of observable phenomena are investigated. A set of axioms of the properties of observation schemes have been proposed. It is worthy noting that the selection of a set of observable contexts plays a crucial role in algebraic testing. This actually corresponds to the determination of an observation scheme in our general theory of testing [22~24].

6. References

[1] Guttag, J., "Abstract data types and the development of data structures", Commun. ACM 20(6), 1977, pp396-404.

[2] Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B., "Initial Algebra Semantics and Continuous Algebras", J. ACM 24(1), Jan.1977, pp68 – 95.

[3] Goguen, J. and Malcolm, G., "A hidden agenda", Theoretical Computer Science 245(1), 2000, pp55-101.

[4] Sannella, D. and Tarlecki, A., "Essential Concepts of Algebraic Specification and Program Development", Formal Aspects of Computing 9(3) , 1997, pp229-269.

[5] Sannella, D. and Tarlecki, A., "Algebraic methods for specification and formal development of programs", ACM Comput. Surv. 31(3es), Article 10, Sept. 1999.

[6] Gonnon, J., McMullin, P. and Hamlet, R., "Data-Abstraction Implementation, Specification, and Testing", ACM TOPLAS 3(3), July 1981, pp211-223.

[7] Bernot, G., Gaudel, M. C., and Marre, B., "Software testing based on formal specifications: a theory and a tool", Software Engineering Journal, Nov. 1991, pp387- 405.

[8] Bouge, L., Choquet, N., Fribourg, L., and Gaudel, M. C., "Test sets generation from algebraic specifications using logic programming", J. of Systems and Software 6(4), 1986, pp343 -360.

[9] Doong, R. K. and Frankl, P. G., "Case studies on testing object-oriented programs", Proc of the symposium on Testing, analysis, and verification, 1991, pp165-177.

[10] Doong, R. K. and Frankl, P. G., "The ASTOOT approach to testing object-oriented programs" ACM TSEM 3(2) , Apr.1994, pp101-130.

[11] 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 TSEM 10(1), Jan. 2001, pp56-109.

[12] 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. 7(3), Jul. 1998, pp250-295.

[13] Orejas, F., Navarro, M, and Sanchez, A., "Implementation and behaviour equivalence: a survey", Proc. of 8th WADT/3rd COMPSAA Workshop, LNCS 655, 1993, pp93-125.

[14] Hennicker, R., "Observational implementation of algebraic specifications", Acta Informatica 28, 1991, pp187-230.

[15] Gaudel, M, C., "Testing can be formal", Proc. of TAPSOF'95, LNCS 915, 1995, pp82-96.

[16] Nivela, P. and Orejas, F., "Initial behavioural semantics for algebraic specifications", Recent Trends in Data Type Specification, LNCS 332, 1988, pp184-207.

[17] Sannella, D. T., and Wirsing, M., "A kernel language for algebraic specification and implementation", Proc. of 1983 International Conference on Foundation of Computation Theory, Borgholm, Sweden, 1983, LNCS 158, pp413-427.

[18] Sannellla, D., and Tarlecki, A., "On observational equivalence and algebraic specification", Journal of Computer and System Sciences 34, 1987, pp150-178.

[19] Bidoit, M. and Hennicker, R., "Behavioural theories and the proof of behavioural properties", Theoretical Computer Science, 165(1), 1996, pp3-55.

[20] Hofmann, M., and Sannella, D., "On obsevioural abstraction and behavioural satisfaction in higher-order logic", Proc. of TAPSOFT'95, LNCS, 1995.

[21] Bergstra J.A., Tucker J.V., "Initial and Final Algebra Semantics for Data Type Specifications: Two Characterization Theorems", SIAM Journal of Computing 12, 1983, pp366-387.

[22] Zhu, H. and He, X., "An Observational Theory of Integration Testing for Component-Based Software Development", Proc. of COMPSAC'2001, October 2001, Chicago, Illinois.

[23] Zhu, H. and He, X., "Constructions of behaviour observation schemes in software testing", Proc. of HASE'2000, Albuquerque, New Mexico, USA, Nov, 2000, pp7-16.

[24] Zhu, H. and He, X., "A methodology of testing high-level Petri nets", Information and Software Technology 44(8), June 2002, pp473-489.