Validating Algebraic Class Testing in Final Algebra

Hong Zhu

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

Abstract

Algebraic class testing is an object-oriented software testing method based on algebraic specification. It tests if a class correctly implements an algebraic specification by checking if the equations of the specification are satisfied. Each test case consists of two ground terms, which corresponds to two sequences of method calls to the class, and a tag indicating if the sequences should generate equivalent results according to the specification. One of the most appealing advantages of algebraic testing is that formal specifications can be used to automatically generate test cases and to determine if two values are equivalent, for example, by using observation contexts. A basic validity requirement for test oracles is that it correctly determines if the equivalence of the values of ground terms is consistent with the equivalence required by the algebraic specification. Unfortunately, for initial algebra semantics, this requirement is not satisfied by the observation contexts. This paper proves that the observation context approach satisfies the validity requirements for the final algebra semantics of algebraic specifications.

Keywords: *Software testing, algebraic specification, initial algebra and final algebra, test oracle, object-oriented software, specification-based testing, automated testing*

1 Motivation

Algebraic class testing is an automated software testing method for testing object-oriented programs at class level based on algebraic specifications [1, 2, 3, 4]. It is based on the observation that each term of a given signature has two interpretations in the context of software testing. First, it represents a sequence of calls to the methods that implement the operators specified in the algebraic specification. When the variables in the term are replaced with constants, such a sequence of calls represents a test execution of the program, where the test case consists of the constants that are substituted for the variables. Second, a term also represents a value, i.e. the result of the execution. Therefore, checking if an equation is satisfied by an implementation on a test case means executing the operation sequences of two terms on both sides of the equation and then comparing the results. If the results are equivalent, the program is correct on this test case; otherwise the implementation has errors. One of the most appealing features of algebraic testing is that formal specifications can be used to automatically generate test cases and to determine if the program produces correct output. A high degree of test automation can be achieved.

Algebraic specification emerged in the 1970s and developed through out the 80s and 90s as a formal method for specifying abstract data types in an implementation-independent style; see, e.g. [5, 6]. In late 1980s, Gaudel *et al.* developed a theory and a method of specification based software testing [7, 8]. Using a set of prototype tools, they demonstrated that algebraic specifications could be used to solve both the problems of automatic test case generation and automatic test oracle and test driver generation. Although it was not directly targeted to class

testing of object-oriented software, their work laid the theoretical foundation of algebraic testing in general. The approach received much attention recently in the context of class testing. Frankl and Doong studied the effectiveness of testing object-oriented programs based on algebraic specifications [1, 2]. They developed an algebraic specification language called LOBAS and a tool called ASTOOT for testing object-oriented programs. They conducted two case studies to assess the practical usability of the method and technique. 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. More recently, Chen and Tse, *et al*. further developed the theory and method of automatic derivation of test oracles based on observation contexts [3, 4]. They raised a number of subtle and serious questions about the foundation of algebraic testing, including the validity problem of the test oracles based on observation contexts.

In this paper, we answer the validity question raised in [4] based on the theory of algebraic specifications. The remainder of the paper is organised as follows. Section 2 provides the preliminaries of the theory of algebraic specifications. Section 3 introduces the basic concepts of algebraic class testing and analyses the assumptions underlying the method. It then discusses the oracle problem and summarises the main result of the paper. Section 4 proves the main theorems of the paper. Section 5 is the conclusion.

2 Preliminaries of Algebraic Specification

The basic idea of algebraic specification is to treat an abstract data type as an algebra, which consists of a collection of sets as the carriers of the algebra and a number of operations on the sets. Each set corresponds to a type and is called a *sort*. An algebraic specification consists of two main parts. The syntax part is called the signature, which defines the name of the operations and sorts of their operands and result values. The semantics part defines the meanings of the operations by a set of axioms represented in the form of equations, or conditional equations. These axioms are the properties that the operations must satisfy. Each equation consists of two terms. It means that the terms are equivalent for any values substituted systematically into the variables on both sides of the equation. The order that the axioms are listed is not important. The following defines the notions and notation as well as the main results about algebraic specifications that are used in this paper. The proofs of the results are omitted. Readers are referred to [9] for a textbook and [10, 11, 12, 13] for surveys.

A *signature* Σ consists of a nonempty set S whose elements are called *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 $<w, s>$, i.e. w is the *arity* of the operator and s is its sort.

Given a signature Σ and disjoint sets V_s, $s\in S$, of Σ-variables[1], the set of Σ-*terms* is inductively defined as follows.

(1)For all constants $\sigma\in\Sigma_{\lambda,s}$, σ is a Σ-term of sort s;
(2)For all variables $v\in V_s$, v is a Σ-term of sort s;
(3)For all operators $\sigma\in\Sigma_{w,s}$, $w = w_1\ldots w_n$, and terms $\tau_1, \ldots\tau_n$ of sorts $w_1, \ldots,w_n$, respectively, $\sigma(\tau_1, \ldots\tau_n)$ is a Σ-term of sort s.

A term is called a *ground* term, if it contains no variable. We write $W_{\Sigma,s}(V)$ to denote the set

[1] The set of variables must be disjoint to the set of operator symbols, too.

of Σ-Terms of a sort s with variables in V and $W_\Sigma(V)$ to denote the set of all Σ-terms with variables in V. We write $W_{\Sigma,s}$ and W_Σ to denote $W_{\Sigma,s}(\varnothing)$ and $W_\Sigma(\varnothing)$, which are ground terms.

A Σ-*equation* consists of two Σ-terms of the same sort and written in the form of $\tau_1=\tau_2$. A *conditional* Σ-*equation* consists of a Σ-equation and a condition in the form of the conjunction of a number of Σ-equations, such as

$$\tau = \tau', \quad if \quad \tau_1 = \tau_1' \wedge ... \wedge \tau_k = \tau_k'.$$

A Σ–algebra A consists of a family $<A_s>_{s\in S}$ of sets called the *carriers* of A, where A_s is the carrier of sort $s\in S$; and for each $<w,s>\in S^*\times S$ and $\sigma\in\Sigma_{w,s}$, there is an operation σ_A of type $<w,s>$, i.e. $\sigma_A: A_{w_1}\times...\times A_{w_n} \to A_s$, where $w = w_1\ldots w_n$, and $w_i\in S$ for all i=1,2,...,n. An operation σ_A of type $<\lambda, s>$ is a constant of sort s, i.e. $\sigma_A\in A_s$, where $\lambda\in S^*$ is the empty string. Such an algebra is a *many-sorted algebra*.

Let $V=\bigcup_{s\in S} V_s$ be a set of Σ-variables, where V and Σ are disjoint. An *assignment* is a mapping φ from V to A, such that for all $v\in V_s$, $\varphi(v)\in A_s$. The value of a Σ-term τ under an assignment φ, written $[\![\tau]\!]_\varphi$, is inductively defined as follows.

(1)For all constants $\sigma\in\Sigma_{\lambda,s}$, $[\![\sigma]\!]_\varphi = \sigma_A$;

(2)For all variables $v\in V_s$, $[\![v]\!]_\varphi = \varphi(v)$;

(3)For all operators $\sigma\in\Sigma_{w,s}$, $w = w_1\ldots w_n$, and terms $\tau_1, \ldots\tau_n$ of sorts $w_1, \ldots,w_n$, respectively, $[\![\sigma(\tau_1,...,\tau_n)]\!]_\varphi = \sigma_A([\![\tau_1]\!]_\varphi,...,[\![\tau_n]\!]_\varphi)$.

Notice that, if a term τ is a ground term, we have that for all assignments φ and φ' in a Σ-algebra A, $[\![\tau]\!]_\varphi = [\![\tau]\!]_{\varphi'}$. This means that a ground term has a fixed unique value in a Σ-algebra A. This value in the algebra A is denoted by $[\![\tau]\!]_A$. When there is no risk of confusion, we omit the subscript and simple write $[\![\tau]\!]$.

A Σ-algebra A *satisfies* a Σ-equation $\tau_1=\tau_2$, written $A\models(\tau_1=\tau_2)$, if $[\![\tau_1]\!]_\varphi = [\![\tau_2]\!]_\varphi$ for all assignments φ in A. A satisfies $\tau = \tau', \ if \ \tau_1 = \tau_1' \wedge ... \wedge \tau_k = \tau_k'$, written $A\models(\tau = \tau', \ if \ \tau_1 = \tau_1' \wedge ... \wedge \tau_k = \tau_k')$, if $[\![\tau]\!]_\varphi = [\![\tau']\!]_\varphi$ for all assignments φ in A that satisfy the condition $[\![\tau_i]\!]_\varphi = [\![\tau_i']\!]_\varphi$, for all i = 1,2,…,k.

Let E be a set of Σ-equations or conditional Σ-equations. A Σ-algebra A is said to be a $<\Sigma, E>$-algebra, written $A\models E$, if for all $e\in E$, $A\models e$.

Given a set E of equations, one can deduct if a pair of terms τ and τ' is equivalent or not by using equational logic; see, e.g. [11]. If equation $\tau=\tau'$ is derivable from the equations in E, we write $E|-\ \tau=\tau'$.

A substitution μ is a mapping from the set W_Σ(V) of Σ-terms to W_Σ(V) that satisfies the condition that

$$\mu(\sigma(\tau_1,\tau_2,...,\tau_n)) = \sigma(\mu(\tau_1),\mu(\tau_2),...,\mu(\tau_n)),$$

for all n-ary operator $\sigma \in \Sigma$. Therefore, a substitution is determined by its restriction to the set of variables. If for all $v \in V_s$, $\mu(v)$ is a ground Σ-term, the substitution is called a ground term substitution.

The equational logic is complete and consistent in the following sense. Let $E \models \tau = \tau'$ to denote the statement *for all algebra A*, $(A \models E \Rightarrow A \models \tau = \tau')$.

Proposition 2.1 The equational logic is consistent and complete, i.e. $E \vdash \tau = \tau' \Leftrightarrow E \models \tau = \tau'$. □

However, equational logic is not decidable. There is no effectively computable algorithm that can determine if a pair of terms can be deducted from a set of equations.

For certain specific types of equation sets, decidability can be achieved. One of the main techniques for achieving decidability is term rewriting.

An abstract *reduction system* is a structure $< A, \rightarrow >$, where A is a non-empty set, $\rightarrow$ is a binary relation on A, called the reduction relation. The relation $\rightarrow$ is often defined as the union of binary relations $\rightarrow_\alpha$ on A, i.e. $\rightarrow = \bigcup_{\alpha \in I} \rightarrow_\alpha$. The transitive and reflexive closure of a binary relation $\rightarrow$ is denoted by $\rightarrow^*$. The equivalent relation generated by $\rightarrow^*$ is called the *convertibility* relation generated by $\rightarrow$, written $\approx_{\rightarrow^*}$. An element $a \in A$ is a *normal form*, if there is no $b \in A$ such that $a \rightarrow b$. An element $a' \in A$ *has a normal form* if $a' \rightarrow^* a$ for some normal form $a \in A$. The reduction relation $\rightarrow$ is *weakly normalising*, if every $a \in A$ has a normal form. It is said *strongly normalising*, if every reduction sequence $a_0 \rightarrow a_1 \rightarrow ...$ eventually must terminate. Strongly normalising is also called *terminating*, or *Noetherian*. The reduction relation $\rightarrow$ has the *unique normal form property*, if for all a, $b \in A$, $a \approx b$ and a, b are normal forms imply that $a = b$. It is called *confluent* or *has the Church-Rosser property*, if for all a, b, $c \in A$, $a \rightarrow^* b$ and $a \rightarrow^* c$ implies that there is $d \in A$ such that $b \rightarrow^* d$ and $c \rightarrow^* d$. A reduction system is said *complete*, or *canonical*, or *uniquely terminating*, if it is confluent and terminating.

Given a signature Σ, a *term rewriting system* is a reduction system on the set of Σ-terms. The collection of reduction relations is defined by a number of *term rewriting rules*, which are ordered pairs (τ, τ') of terms in $W_\Sigma(V)$, usually written as $\tau \rightarrow \tau'$. Two conditions are imposed on the terms.

(1) The left-hand side τ is not a variable;
(2) The variables in the right-hand side τ' are already contained in τ.

A *Context* is a term containing one occurrence of a special symbol □, denoting an empty place. It is generally denoted by $C[\ ]$. If $\tau \in W_\Sigma(V)$ and τ is substituted in □, the result is $C[\tau]$; τ is said to be a *subterm* of $C[\tau]$.

A term rewriting rule r: $\tau \rightarrow \tau'$ determines a binary reduction relation $\rightarrow_r$ on the set of Σ-terms such that $C[\mu(\tau)] \rightarrow_r C[\mu(\tau')]$ for all context $C[\ ]$ and all substitutions μ. The sub-term $\mu(\tau)$ of left-hand side in the term $C[\mu(\tau)]$ is called a *redex* (from 'reducible expression'). A redex $\mu(\tau)$ may be replaced by its 'contractum' $\mu(\tau')$ inside the context $C[\ ]$; this gives rise to a reduction step, or one-step rewriting $C[\mu(\tau)] \rightarrow_r C[\mu(\tau')]$.

Given an algebraic specification $<\Sigma, E>$, the set E of equations can be considered as a set of

rewriting rules, where an equation $\tau = \tau'$ is considered as a rule $\tau \rightarrow \tau'$. A conditional equation

$$\tau = \tau', \ \ if \ \ \tau_1 = \tau_1' \wedge ... \wedge \tau_k = \tau_k'$$

is considered as a conditional term rewriting rule:

$$\tau \rightarrow \tau', \ \ if \ \ \tau_1 \approx \tau_1' \wedge ... \wedge \tau_k \approx \tau_k' \ .$$

The term rewriting system $\rightarrow_E$ derived from an algebraic specification <Σ, *E*> as above has the following property.

Proposition 2.2
(1) For all Σ-terms τ, $\tau' \in W_\Sigma(V)$, $(\tau \rightarrow^* \tau') \Rightarrow E \ |\!- (\tau = \tau')$.
(2) For all Σ-terms τ, $\tau' \in W_\Sigma(V)$, $(\tau \approx_E \tau') \Rightarrow E \ |\!- (\tau = \tau')$.

The inverses of the above statements are not necessarily true. If the rewriting system has the property that $(\tau \approx_E \tau') \Leftrightarrow E \ |\!- (\tau = \tau')$, the term rewriting system is *complete*. If the term rewriting system derived from an algebraic specification is canonical, we say that the specification is canonical.

Given an algebraic specification <Σ, *E*>, there may exist a collection of Σ-algebras that satisfy the equations. Such an algebra is called a *model* of the specification. The semantics of algebraic specification language determines which of the models or which subset of the models is what we mean by correct implementation.

The *loose* semantics of a specification is the whole class of models satisfying the axioms. A loose semantics may contain trivial implementations as 'correct' implementation. Such models cannot be easily ruled out using pure or even conditional equations. A solution to this problem is, therefore, to use a more powerful logic, such as inequations, in the specifications.

The *initial algebra* semantics defines the semantics of an algebraic specification to be the initial algebra of the models. An algebra *A* is an initial algebra in a collection *C* of algebras, if for all *A*'∈*C*, there is a unique homomorphism φ from *A* to *A*', where a mapping from *A* to *A*' is a *homomorphism*, if

(1)for all $a \in A_s$, $s \in S$, we have that $\varphi(a) \in A_s'$; and

(2)for all $\sigma \in \Sigma_{w,s}$, $w = w_1...w_n$, and $a_1, a_2, ..., a_n$, $a_i \in A_{w_i}$, $w_i \in S$, *i*=1,2,…,*n*, we have that

$$\varphi(\sigma_A(a_1, ..., a_n)) = \sigma_{A'}(\varphi(a_1), ..., \varphi(a_n))$$

If an initial algebra exists in a collection of models, it is unique up to isomorphism. Initial algebra is characterised by the fact that all elements of its carriers are the interpretation of a ground term (known as the *no junk* property) and that it satisfies only those ground equations that hold in all models of the algebraic specification (known as the *no confusion* property). The initial algebra can be understood through the equality of terms in the algebra as follows [14].

Proposition 2.3. In the semantics of initial algebra, two ground terms denote different objects unless it can be proved from the stated axioms that they denote the same object. Formally, let <Σ, *E*> be an algebraic specification, and *A* be the initial Σ-algebra of the specification *E*.

$A \models (\tau = \tau') \Leftrightarrow E \ |\!- (\tau = \tau')$ □

Notice that, since equational logic is consistent and complete, Proposition 2.3 implies that $A \models (\tau = \tau') \Leftrightarrow E \models (\tau = \tau')$. This is the so called 'no confusion' property mentioned above.

Final algebra is another frequently used semantics of algebraic specifications. An algebra B is a *final* algebra in a collection C of models, if B is not a unit algebra[2] and for all models B' in the collection there is a unique homomorphism from B' to B. The final algebra can also be understood through the inequality of terms as follows [14].

Proposition 2.4. In the semantics of final algebra, two ground terms of the same sort denotes the same object unless it can be proved from the stated axioms that they denote different objects. Formally, let $<\Sigma, E>$ be an algebraic specification, and A be the final algebra of the specification.

$A \models (\tau = \tau') \Leftrightarrow (\tau = \tau')$ is consistent with E. □

The difference between the initial algebra and final algebra can be illustrated by the bank account example given later in Example 3.3 given in section 3.

3 Algebraic Class Testing And The Oracle Problem

In this section, we give a brief introduction to the basic ideas of algebraic class testing and discuss the applicability of the method by analysing the assumptions underlying the method. We then discuss the oracle problem in algebraic class testing.

3.1 Basics of algebraic class testing

Figure 3.1 below depicts a conceptual model of the testing activities involved in algebraic class testing. Algebraic class testing is a specification-based method. Test cases are generated from the formal algebraic specifications of the classes. These test cases are represented in terms of the symbols defined in the algebraic specification. To understand these symbols in the context of the implementation represented as a set of classes, the relationship between the specification and its implementation must be obtained. In particular, we need a mapping between the operations and sorts given in the specification and the class identifiers and the names of the attributes and methods. The class interface can also be considered as a signature. Such a mapping is then a *signature morphism*. To execute the classes on the test cases and to automatically check the correctness of the test executions, test drivers and oracle must be generated from the implementation. Finally, test cases, test drivers and test oracle must be put together and executed to produce a test report.

[2] A many-sorted algebra U is a unit algebra, if for all sorts s, the carrier U_s is a singleton set.

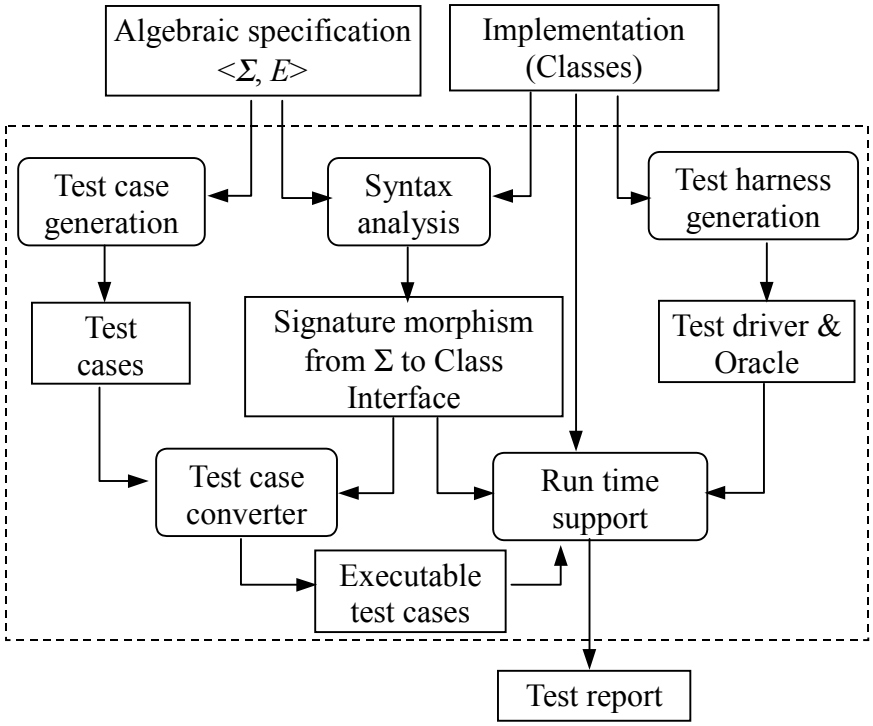


Figure 3.1. A conceptual model of the activities in algebraic class testing

Although abstract data type is one of the fundamental concepts underlying object-orientation, there are number of significant and subtle differences between algebras and classes.

(1) State variables and side effects

A class consists of a set of attributes and a set of methods. The attributes represent the state of an object of the class. The methods change the state according to its current state. A method may return values according to the parameters and the state as well. The involvement of states in a method is fundamentally different from the functional nature of algebraic operations. To resolve this difference, the set of all possible states of an object of the class is regarded as the carriers of a sort. Here, a class in an object-oriented program corresponds to a sort in an algebraic specification. When a method is executed, the state of the object before the execution is considered as one of the operands of the corresponding operation and the state of the object after executing the method is considered as the result of the operation. This requires that every operation can have only one occurrence of the sort that represents the state in the arity of the operation. If a method involves another object of the same class, its arity does contain a second occurrence of the class' sort. However, it must be made clear in the definition of the signature of the operation that which one stands for the state and which one for other objects. Doong and Frankl proposed a very nice syntactic notation for this purpose in their algebraic specification language LOBAS specially designed for algebraic class testing. The notation assumes that the state sort is the first sort in the arity and uses a dot and the method name as the delimiter between the first sort and the rest of the arity. For example, for the `put` operator of natural number queues with arity

```
put : Queue × Nat → Queue;
```

its type declaration in LOBAS takes the first sort and the result sort `Queue` as the default and omitted. Hence, it is declared as follows.

```
put(x:Nat).
```

An expression that involves the `put` operator, such as `put(create, 1)`, is represented in the LOBAS as follows.

```
create.put(1)
```

This notation actually is an extension of the surfix representation of expressions, hence the sequence of method calls in an expression is the same as the writing order from left to right.

Side effects may occur in the implementation of a class in two forms. Firstly, methods in a class are not functions by the nature that they have side effects on the states in addition to the return values. For example, a class that implements queues may well contain a method that returns the first value and deletes the element from the queue. Such a method cannot be specified algebraically. A solution to this problem is to split the functionality of the method into two operations, one for deleting the first element and one for returning the first element. A consequence is that there are no one-one correspondences between the signature of the algebraic specification and the class interface. It makes the mapping from a term in algebraic specification into a sequence of method calls less straightforward. The researchers on algebraic class testing assumed that such situations do not occur.

Secondly, side effects also occur in the form of changing the state of the method's argument object. Such side effects cannot be specified by algebraic specifications. Therefore, algebraic testing is not applicable to classes with such side effects.

(2) System structure

An object-oriented program usually consists of a number of classes. The class under test cannot be executed in isolation without 'importing' its supporting classes, which may be the classes of its attributes, the classes of the parameters of the methods, the result class of a method, or a class used to implement the methods. Obviously, such importing/ supporting relationship is a pre-order. At the lowest level, there are a number of pre-defined classes, such as those support input/output facilities, and basic data types such as 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.

By considering classes as sorts, one would expect that an algebraic specification should be decomposed into units of similar relationship. Unfortunately, existing theories of algebraic specification do not guarantee such a relationship.

In algebraic specifications, one of the most important system-building operations is *enrichment* or *extension*; see e.g. [12, 13]. A unit of algebraic specification enriches or extends other unit(s) to compose units together and to build new units on the base of existing ones. The semantics of such an extension is to put all the sorts, operations and their axioms together. The specification that extends another may have additional operations and/or axioms defined for the sort(s) already defined in the existing specification units. Therefore, enrichment is more like the inheritance relationship between classes. In order to simplify the relationship between specification and implementation, an algebraic specification should be decomposed into units that resemble the importing / supporting relationship between classes. Therefore, each unit in the specification should have a main sort and a number of supporting sorts. The axioms of the unit should not modify the semantics of the supporting sorts that are defined in other units. In testing a class, only the axioms for the main sort need test rather than the axioms of supporting sorts. Supporting classes are assumed to be correct or have

been adequately tested. In particular, a basic class is assumed to have been correctly implemented by the system, and correctly selected for the specification unit 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,

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

An algebra A is a correct implementation of an observable sort s, if for all ground terms τ and τ' of sort s,

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

As we will show later, the distinction between a main sort and the supporting sorts not only decides which axioms are to be checked, it also plays a significant role in the derivation of test oracles. The importing/ supporting relation on the sorts / classes must have the following properties.

(1) The importing / supporting relation on the sorts/classes is a pre-order $\prec$ on the sorts so that $s_1 \prec s_2$ means that s_1 is a sort that supports sort s_2;

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

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

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

Definition 3.2 (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, $w_i \neq c$. 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 the operator σ can appear in at least one normal form of ground terms.[3]

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 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 there is at least one $i \in \{1, 2, \ldots, n\}$, such that $w_i = c$, $s \neq c$ and $s \prec c$. $\square$

Obviously, an operator in a canonical algebraic specification is a creator, or constructor, or transformer, or observer. Moreover, it can be only one of these types.

The axioms of an algebraic specification should also preserve the pre-order of 'support' relation in the following sense.

[3] We also assume that the LOBAS notation is used, 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 class.

(4) For all sorts s, $s' \in \Sigma$, $s' \prec s$ and s' is directly support s (formally, $\neg\exists s'' \in \Sigma.(s' \prec s'' \wedge s'' \prec s)$), there is an observer σ of sort s such that $\sigma : w_1 \times \ldots \times w_n \rightarrow s'$;

(5) For all conditional equations ($\tau = \tau'$, if $\tau_1 = \tau_1' \wedge \ldots \wedge \tau_k = \tau_k'$), for all i = 1, 2, ..., k, $s_i \prec s$ or s_i is an observable sort, where s_i is the sort of τ_i and τ_i', s is the sort of τ and τ'.

Definition 3.3 (Well structured specification)

A specification <Σ, *E*> is *well structured with respect to* $\prec$, if it satisfies properties (1) ~(5). □

3.2 The oracle problem

A basic problem in algebraic testing is to decide if two values of an abstract data type are equivalent. The following example shows that this is not a trivial problem.

Consider the circular array implementation of queues described in Example 3.1.

Example 3.1 (Circular array implementation of queues)

As depicted in Figure 3.2, a circular array implementation of queues contain an array `A`, two integer type variables `Head` and `Tail` that give the indices of the head element in the queue and the tail of the queue, i.e. first available cell in the array for next element.

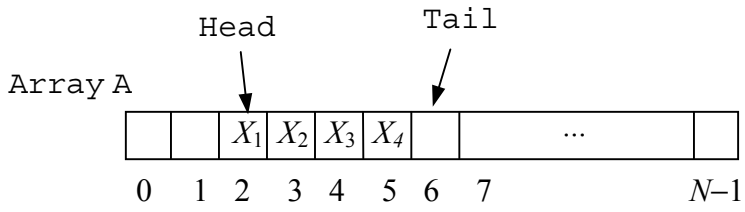


Figure 3.2 Circular array implementation of queue

The operations on queues are implemented as follows.

`Create`: creates a new queue and initialises it with `Head` := 0, `Tail`:=0, and all the cells of the array `A` to be 0.

`Put(x)`: checks if `(Tail+1) mod N = head`; if yes, it means the queue is full, otherwise, assign `x` to `A[Tail]` and then increase `Tail` by `1`. Here, increase `X` by `1` means that `X := (X + 1) Mod N`.

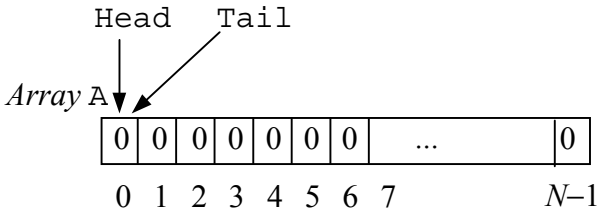
`Get`: checks if `Head = Tail`; If yes, it returns an error message that the queue is empty, otherwise increase `Head` by `1`.

`Front`: returns the value in the cell `A[Head]`.

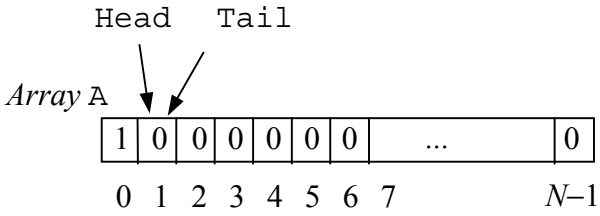
`Is_empty`: returns `true` if `Head=Tail`; otherwise it returns `false`.

`Length`: returns the number of elements in the queue, i.e. if `Tail` ≥ `Head` then return `Tail` – `Head` else return `Tail` + `N` – `Head`.

Now consider the terms `Create` and `Create.Put(1).Get`. They should be equivalent because both of them are empty queues. However, as shown in Figure 3.3, their internal representations are quite different, i.e. not identical. □



(a) The internal representation after executing `Create`



(b) The internal representation after executing `Create.Put(1).Get`

Figure 3.3 Internal representations of circular array implementation of queues

This example shows that determining the equivalence of two values of an abstract data type may not be so straightforward as comparing the values of the state variables of a class.

Let τ be any ground term, the value of the term in an implementation is denoted by $[\![\tau]\!]$. A test oracle for algebraic testing can be defined as a binary relation $\approx$ on the values. There are two validity requirements for any oracle of algebraic testing.

(A) For any correct implementation *A* of a specification *E*, for any two terms τ_1 and τ_2, τ_1 is equivalent to τ_2 according to the specification *E* if and only if $[\![\tau_1]\!] \approx [\![\tau_2]\!]$.

(B) For any incorrect implementation *A* of a specification *E*, there are at least two terms τ_1 and τ_2 such that:

(1) τ_1 is equivalent to τ_2 according to the specification *E*, but $[\![\tau_1]\!] \approx [\![\tau_2]\!]$ is not true; or

(2) τ_1 is not equivalent to τ_2 according to the specification *E*, but $[\![\tau_1]\!] \approx [\![\tau_2]\!]$ is true.

Notice that whether a given test oracle is valid depends on what does it mean by an implementation is correct with respect to a specification. In other words, it depends on the semantics of the specification.

Three approaches have been proposed in the literature to develop or to derive a test oracle [2].

A. Inclusion of ≈ in the Formal Specification.

It is often possible to produce a recursive definition of the ≈ by a binary operation _ == _ : $s \times s \rightarrow Bool$ in the algebraic specification such that $x == y$ if and only if $[\![x]\!] \approx [\![y]\!]$. Therefore, when the specification is canonical, the defining axioms of == can be considered as an abstract definition of the algorithm for ≈. An oracle can be derived from the axioms automatically. It should be noticed that such an oracle would inevitably use the implementations of other operators defined in the class. Errors in the implementation of these operators will propagate to the test oracle ≈. Such error propagation can help detecting the

errors as well as masking the errors. In other words, the validity requirement B of such oracles cannot be guaranteed although the validity requirements A can be satisfied provided that the specification of the oracle is correct.

B. Implementation of ≈ as a Part of the Class.

The approach A can be viewed as writing the oracle at specification level. Another approach to the oracle problem is to write the code at implementation level. A method `EQN(x:S):Bool` can be written for each class `S` that returns `true` if and only if the object `x`'s value is equivalent to the value of the object. Generally speaking, such a method can be implemented correctly with high accuracy if sufficient attention is paid to the representation of the abstract data type. However, like writing any program code, it could be error-prone no matter if it is at specification level or at programming level. Therefore, whether the oracle satisfies the validity requirements A and B depends on the correctness of the code that implements the oracle.

C. Reduction of ≈ to Preliminary Classes

Both of the above approaches involve manually writing code for the test oracle. Hence, they are less automatic. The third approach can achieve a very high degree of automation. To illustrate the basic idea of the approach, consider the following example.

Example 3.2 Two natural number queues q_1 and q_2 should be equivalent, if we apply the following operations to them and generate equivalent results:

```
length(x):                  length(q1) = length(q2);
front(x):                front(q1) = front(q2);
is_empty(x):          is_empty(q1) = is_empty(q2);
front(pop(x)):           front(get(q1)) = front(get(q2));
is_empty(pop(x)):        is_empty(get(q1)) = is_empty(get(q2));
…
front(get^k(x)):         front(get^k(q1))=front(get^k(q2));
is_empty(get^k(x)):      is_empty(get^k(q1)) = is_empty(get^k(q2));
…
```

where $get^1(x) = get(x)$, $get^{k+1}(x)=get(get^k(x))$. □

Notice that, the results of applying the operations given in Example 3.2 are natural numbers. We should be able to determine the equivalence between two natural numbers if the natural numbers are implemented by a basic class of the programming language and there is a correct implementation of the = operation. By applying such operations, we actually reduced the problem of determining the equivalence between two values of an abstract data type to the equivalence of values in a basic data type. The equivalence between values of a basic type can be easily and effectively decided. These operations are called *observation contexts*.

Researchers on algebraic testing are in favour of this approach because it can achieve a high degree of automation in oracle derivation. However, similar to the first two approaches, the correctness of such test oracles depends on the correctness of the implementation of the operations. In other words, the errors in the implementation of the operations may propagate to the oracle. Therefore, it may mask the errors as well as help detecting errors. An example of error masking in an incorrect implementation of the natural number stacks is given by Bernot, Gaudel and Marre [7]. In other words, the validity requirements B cannot be guaranteed by this approach. One would expect that the observation context approach satisfies the validity requirement A. Unfortunately, it is not true. Consider the following counterexample, which originates from Chen, Tse, *et al*. [4].

Example 3.3 (Bank accounts) Consider the following algebraic specification of bank accounts.

```
Spec Account
     Sorts: Account, Money, String;
     Operations:
          overdrawn: → Money;
          new(_): String → Account;
          _.name: Account → String;
          _.addr: Account → String;
          _.bal: Account → Money;
          _.setAddr(_): Account × String → Account;
          _.credit(_) : Account × Money → Account;
          _.debit(_): Account × Money → Account;
     Variables:
          S: String; A: Account; M: Money;
     Axioms:
          A1: new(S).name = S
          A2: new(S).addr = nil
          A3: new(S).bal = 0
          A4: A.credit(M).bal = A.bal + M
          A5: A.debit(M).bal = A.bal - M, if A.bal≥M
          A6: A.debit(M).bal=overdrawn, if A.bal<M
          A7: A.setAddr(S).bal = A.bal
          A8: A.credit(M).addr = A.addr
          A9: A.debit(M).addr = A.addr
          A10:A.setAddr(S).addr = S
          A11:A.credit(M).name = A.name
          A12:     A.debit(M).name = A.name
          A13:A.setAddr(S).name = A.name
end
```

Now consider the following two terms of the sort `Account`.

u_1: `new('John').setAddr('2 Univ Drive').credit(1000). debit(200)`

u_2: `new('John').setAddr('2 Univ Drive').credit(800)`

Chen and Tse *et al.* noticed that the equivalence of the terms u_1 and u_2 couldn't be proved in equational logic from the axioms. However, even if the implementation is correct, the difference between the two terms cannot be detected by observation contexts for the following reasons. There are three observers of the sort `Account`: `_.name`, `_.addr`, and `_.bal`. It is easy to see that for all observable context oc of sort `Account`, u_1.oc == u_2.oc. In other words, $u_1 \approx u_2$. □

Chen and Tse, *et al*. correctly pointed out that the conflict between $u_1 \neq u_2$ and $u_1 \approx u_2$ in the above example is a fundamental problem of the observation context approach to test oracle.

3.3 Main contribution of the paper

A clue to the problem is that whether two terms should be consider as equivalent depends on the semantics of the specification. For the bank account example given in Example 3.3, in the final algebra semantics, we have that $u_1 = u_2$ because the equation $u_1 = u_2$ is consistent with the axioms. However, in the initial algebra semantics, we have that $u_1 \neq u_2$ because we cannot derive the equation from the axioms in equational logic. This difference between initial and final semantics has a significant impact on the validity of test oracles. The above example

actually shows that the test oracle based on observation contexts is invalid, hence not suitable, for testing software against the initial semantics of algebraic specifications.

The main result of the paper to be proved in the next section is that the oracle based on observation contexts is valid for testing object-oriented software against an algebraic specification if and only if the semantics of the algebraic specification is the final algebra.

4 Validation of Observation Context Oracles

This section first review the notion of observation context proposed by Gaudel *et al.* [7, 8] and further developed by Chen and Tse, et al. [4]. It then proves that test oracles based on observation contexts are valid for testing final algebras.

4.1 Observable equivalence

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 support relation between the sorts.

Definition 4.1 (Observable context)
An *observable context oc* of sort *c* is a context *oc*[ ] whose □ place is of sort *c* and the result sort 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. ….oc_n of a sequence of observable contexts oc_1, oc_2, …, oc_n , where _.oc1: $c \rightarrow s_1$, _.oc$_i$: $s_{i-1} \rightarrow s_i$, for all i =2,…,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:

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

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

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 4.1 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. □

For example, assume that `Nat` is implemented by a pre-defined class *integer* and there is an operation for test equivalence between two integer values. `Nat` is, then, an observable sort. `length(x)`, `is_empty(x)` and `front(x)` are observers of `Queue`. The operations

`length(x)`, `front(x)`, `is_empty(x)`, `front(get`^{k}`(x))` and `is_empty(get`^{k}`(x))`, for all k =1, 2, …, given in Example 3.2 are observable contexts of sort `Queue`. Since there are only three sorts in the specification of natural number queues and `Nat` and `Bool` support `Queue`, we have that the queue specification and its implementation is well structured.

Definition 4.2 (Observational equivalence of terms)
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 [4].

Lemma 4.2 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 4.3 (Subsume relationship theorem)

Given a canonical specification <Σ, *E*>, for all ground terms τ and τ' of same sort, we have that $E \vdash \tau = \tau'$ implies that $\tau \sim_{obs} \tau'$; but the converse is not always true. □

The specification of bank account given in Example 3.3 is a counterexample of the converse of Lemma 4.3.

4.2 Characteristic theorem

The importance of Lemma 4.3 is that it proves that observational equivalence is not always the same as the equivalence relation in the initial algebra; see Proposition 2.3. We now prove that observational equivalence is the same as the equivalence relation in the final algebra.

Lemma 4.4 The relation $\sim_{obs}$ is an equivalence relation on the set W_Σ of ground Σ-terms.

Proof. The statement follows Lemma 4.2. 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 4.2, 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 4.2, 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 4.2, 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 4.2, 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 [4] is not congruent to the operations in the specification $<\Sigma, E>$, because the context $C[...]$ can be a constructor rather than an observer.

Definition 4.3 (E-congruence)

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

$$\tau = \tau' \text{, if } (\tau_1 = \tau_1') \wedge (\tau_2 = \tau_2') \wedge ... \wedge (\tau_k = \tau_k')$$

and for all assignments φ in the algebra A, $[\![\tau]\!]_\varphi \sim [\![\tau']\!]_\varphi$, if $([\![\tau_1]\!]_\varphi \sim [\![\tau_1']\!]_\varphi) \wedge ([\![\tau_2]\!]_\varphi \sim [\![\tau_2']\!]_\varphi) \wedge ... \wedge ([\![\tau_k]\!]_\varphi \sim [\![\tau_k']\!]_\varphi)$.

Theorem 4.2 (E-congruence theorem)

Given a well structured specification $<\Sigma, E>$. 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

$$\tau = \tau' \text{, if } (\tau_1 = \tau_1') \wedge (\tau_2 = \tau_2') \wedge ... \wedge (\tau_k = \tau_k')$$

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 3.3, for all i = 1, 2, …, k, s_i is observable. Therefore, by Lemma 4.2, $\mu(\tau_i) \sim_{obs} \mu(\tau'_i) \Leftrightarrow E|-\mu(\tau_i) = \mu(\tau_i')$. Thus, we have that $E|-\mu(\tau) = \mu(\tau')$. 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

$$\mu(\tau_1) \sim_{obs} \mu(\tau_1') \wedge ... \wedge \mu(\tau_k) \sim_{obs} \mu(\tau_k') \Rightarrow E|-\mu(\tau_1) = \mu(\tau_1') \wedge ... \wedge E|-\mu(\tau_k) = \mu(\tau_k').$$

Therefore, in equational logic, we have that $E|-\mu(\tau) = \mu(\tau')$. By Lemma 4.3, we have that $\mu(\tau) \sim_{obs} \mu(\tau')$. □

Corollary of Theorem 4.2**.**

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

Proof. Let B be the final algebra of all $<\Sigma, E>$-alengras. By the property of final algebra [14], 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 $\sim_{obs}$ is an E-congruence and not

unit as proved in Theorem 4.2. □

Theorem 4.3 (Characteristic theorem)

The term algebra $W_\Sigma/\sim_{obs}$ 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, $\tau \sim_{obs} \tau'$ implies $B|=\tau=\tau'$. The following proves that for all ground terms τ and τ', $B|=\tau=\tau'$ implies that $\tau \sim_{obs} \tau'$. Let τ, $\tau' \in W_{\Sigma,s}$ 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, we have that 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 _.*ocs* be any primitive observable context sequence. We have that $[\![\tau]\!]_B.[\![ocs]\!]_B = [\![\tau']\!]_B.[\![ocs]\!]_B$. Thus, $[\![\tau.ocs]\!]_B = [\![\tau'.ocs]\!]_B$, or equivalently, $B|= \tau.ocs=\tau'.ocs$. Since the sort of the terms $\tau.ocs$ and $\tau'.ocs$ are observable, by the proof (1) above, we have that $\tau.ocs$ and $\tau'.ocs$ have the identical normal forms.

Therefore, by Lemma 4.2, in both cases, $\tau \sim_{obs} \tau'$. □

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 4.4 (Observably equivalent objects)

Two objects a_1 and a_2 of sort s are *observably equivalent*, written $a_1 \approx_{obs} a_2$, 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, $a_1.oc \approx_{obs} a_2.oc$, 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', $\tau \sim_{obs} \tau$' implies that $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$;

(2)*Nonequivalence criterion*: For all ground terms τ and τ', not ($\tau \sim_{obs} \tau$') 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', $\tau \sim_{obs} \tau' \Leftrightarrow [\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$.

First, note that A is isomorphic to $W_\Sigma/\sim_{obs}$. Let θ be the isomorphism between these two algebras. Second, note that for all ground terms τ, $[\![\tau]\!]_A = \theta([\tau]_\sim)$, where $[\tau]_\sim$ is the equivalence class of t under the relation $\sim_{obs}$.

For all ground terms τ and τ', we have that $\tau \sim_{obs} \tau' \Leftrightarrow [\tau]_\sim = [\tau']_\sim \Leftrightarrow \theta([\tau]_\sim) = \theta([\tau']_\sim) \Leftrightarrow [\![\tau]\!]_A = [\![\tau']\!]_A \Leftrightarrow A \models \tau = \tau' \Rightarrow [\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$. Thus, $\tau \sim_{obs} \tau' \Rightarrow [\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$.

To prove that $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!] \Rightarrow \tau \sim_{obs} \tau'$, consider the sort s of the terms τ and τ'. If s is observable, by Definition 3.1 and Definition 4.4, we have that $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!] \Rightarrow A \models \tau = \tau'$. By Definition 3.1 and Definition 4.2, we have that $\tau \sim_{obs} \tau'$. If the sort s is not observable, by Definition 4.4, $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]$ implies that that for all primitive observable context sequences ocs, $A \models \tau.ocs = \tau'.ocs$. Since the sort of the terms $\tau.ocs$ and $\tau'.ocs$ is observable, we have that $E \vdash \tau.ocs = \tau'.ocs$. By Definition 4.2, we have that $\tau \sim_{obs} \tau'$. □

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 $<\Sigma, E>$, 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 $\tau \sim_{obs} \tau'$ is required by the specification, the test oracle reports that $[\![\tau]\!]_A \approx_{obs,A} [\![\tau']\!]_A$;

(2) if $\tau \not\sim_{obs} \tau'$ 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 [1~4]. This paper proves that the conclusions one can draw are only valid in final algebra semantics.

5 Conclusion

In this paper, we formally proved that the test oracle $\approx_{obs}$ based on observation contexts satisfies the validity requirements for correct implementations of well-structured canonical algebraic specifications if only if the semantics of the specification is the final algebra. Otherwise, the validity requirement is not necessarily satisfied.

A problem for further research is how to determine if the observation context oracle is valid when given an algebraic specification. By the characteristic theorem proved in this paper, it is obvious that if the initial algebra of the specification is isomorphic to the final algebra, the observation context oracle is valid. Such a specification is semantically complete in the sense that for all pairs of ground terms τ_1 and τ_2, equation $\tau_1 = \tau_2$ is consistent with the specification if and only if it can be derived from the axioms in equational logic. Therefore, the problem can be transformed into the problem how to determine the semantic completeness of algebraic specifications. The following questions are of particular importance from practical point of view. Firstly, is semantic completeness decidable? Secondly, is there a systematic

(ideally computable) method that enables us to derive a semantically complete specification from any given specification?

References

[1] Doong R. K. and Frankl, P. G. (1991), Case studies on testing object-oriented programs; Proceedings of the symposium on Testing, analysis, and verification, 1991, pp165 – 177.

[2] Doong R. K. and Frankl, P. G. (1994), The ASTOOT approach to testing object-oriented programs; ACM Trans. Softw. Eng. Methodol. Vol. 3, No. 2, (Apr.), pp101 – 130

[3] Chen, H. Y. Tse, T. H. Chan F. T. and Chen T. Y. (1998), 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.

[4] Chen, H. Y., Tse T. H. and Chen, T. Y. (2001), TACCLE: a methodology for object-oriented software testing at the class and cluster levels; ACM Trans. Softw. Eng. Methodol. Vol. 10, No. 1 (Jan. 2001), pp56 – 109.

[5] Guttag, J. (1977), Abstract data types and the development of data structures; Commun. ACM 20, 6 (Jun.), pp396 – 404.

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

[7] Bernot, G., Gaudel, M. C., and Marre, B., (1991), 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., (1986), Journal of Systems and Software, Vol. 6, No. 4, pp343-360.

[9] Ehrig, H. and Mahr, B. (1985), Fundamentals of Algebraic Specification: Vol 1. Equations and Initial Semantics, Springer-Verlag, Berlin.

[10] Dershowitz, N. and Jouannaud, J.-P. (1990), Rewrite systems, In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, volume B, pages 243--320. Elsevier, 1990.

[11] Klop, J.W. (1992), Term rewriting systems, In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, Handbook of Logic in Computer Science, volume 2, pp1-116, Oxford University Press, New York, 1992.

[12] Sannella, D. and Tarlecki, A., (1997), Essential Concepts of Algebraic Specification and Program Development, Formal Aspects of Computing, Vol. 9, No. 3, pp229-269.

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

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