Testing Java Components Based on Algebraic Specifications

Bo Yu[(1)], Liang Kong[(1)], Yufeng Zhang[(1)], and Hong Zhu[(2)]
(1) Department of Computer Science, National University of Defence Technology, Changsha, China

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

Abstract

This paper presents a method of component testing based on algebraic specifications. An algorithm for generating checkable test cases is proposed. A prototype testing tool called CASCAT for testing Java Enterprise Beans is developed. It has the advantages of high degree of automation, which include test case generation, test harness construction and test result checking. It achieves scalability by allowing incremental integration. It also allows testing to focus on a subset of used functions and key properties, thus suitable for component testing. The paper also reports an experimental evaluation of the method and the tool.

1. Introduction

Component technology has become a key element in the development of large and complex software systems [1, 2, 3]. It has shifted development focus from design and coding to requirements analysis, integration and testing [4, 5, 6, 7]. This paper is concerned with testing components.

1.1 Problems in component testing

Component testing imposes challenges to existing software testing techniques. As reported in [5], a particular difficulty of component testing is the lack of test bench on which components can be executed. The developers thus struggled to test components that have no user interface such as server side components. Consequently, developers have to spend as much time in writing test harness as to develop the component itself [5]. This results in excessive overhead.

From component users' point of view, component testing is even more difficult and problematic [8, 9]. Components are usually delivered as executable code without the source code and design information. Moreover, the executable code usually contains no instrumentation [10]. Thus, component users have very limited ability to control and observe the behaviour of the component under test [11]. Consequently, white-box testing techniques are not applicable to users' testing of software components. In recent years, techniques and methods have been advanced for including code in commercial-off-the-shelf (COTS) components for self-testing; e.g. [12]. However, they are yet to be adopted by COTS producers. Therefore, currently users' component testing has to be specification- based.

1.2 Approach to the problems

In addition to manual component testing methods, researchers on automated testing have explored the uses of formal specifications such as design-by-contract [13] and state transition diagrams [14]. These methods are capable of automatic generation of test cases w.r.t. certain adequacy criteria. However, they offer little support to checking the correctness of test results automatically. As argued in [15], testing based on algebraic specifications (which is called *algebraic testing* in the sequel) is a promising approach.

Algebraic testing techniques were proposed in early 1980s [16]. The theory and method developed from testing abstract data types in procedural languages [17] to testing classes and class clusters in object-oriented software [18,19,20,21,22]. The main advantage of algebraic testing is its full automation of testing process, including test case generation, test harness construction, and test result checking. However, although its theoretical foundation is applicable to component testing, the techniques for the implementation of automated testing tools have to be adapted. For example, in testing a component that consists of a number of classes and data types and depends on a number of other components, it is not practical to require the availability of axioms of all these constituent entities. It is often that only their interfaces are known.

This paper further develops algebraic testing techniques to solve this problem. It also allows testing to focus on a subset of the properties or functions of the component under test. This is particularly important because components are often designed for a broad applicability, but they are often only used on a subset of provided functions [13]. The paper presents a prototype automated testing tool called CASCAT for testing of Java Enterprise Beans and report the results of experiments in the evaluation of its effectiveness.

The remainder of the paper is organised as follows. Section 2 describes an algebraic specification language CASOCC. Section 3 presents the test case generation algorithm. Section 4 presents the prototype testing tool

CASCAT. Section 5 reports the results of experimental evaluation of the testing method. Section 6 concludes the paper by a comparison with the related work and a discussion of future work.

2. Specification Language CASOCC

Algebraic specification (AS) emerged in the 1970s [23]. In the past three decades, it has developed into a mature formal method [24]. In general, an AS consists of two parts $<\Sigma, E>$, where Σ is the *signature* of the algebra that defines a collection of *sorts* and *operators*; E is a set of axioms in the form of conditional equations. The axioms define the semantics of the operators.

In the specification of abstract data types, a sort represents a data type; operators represent the operations on the data type and constants, which are 0-ary operators. For OO software, a sort represents a class and the operators represent methods of the class. The attributes are assumed to be accessed through getter and setter methods. These interpretations of AS must be modified in order to test software components. This section presents a language for algebraic specification of software components and to support automated component testing.

2.1 Syntax and overall structure

A software component is a 'unit of composition with contractually specified interfaces and context dependencies only. It can be deployed independently and subject to composition by third parties' [2]. The interface of a component typically contains two types of information: (a) the functionality that the component provides; (b) the functionality that the component requires. Modern component models define the syntax for specifying such information to enable components reused across organisations and created a COTS component market. However, industrial standards of component models rarely specify the semantics of the functionality provided and required by a component. The language CASOCC is designed to provide a vehicle to specify such semantics and at the same time to support automatic component testing.

The name of the language CASOCC stands for Common AS of Components and Classes. Here, the word 'common' has two meanings. First, the language does not distinguish software components from classes or data types so that it can be applied to all of these types of software entities, which often occur at the same time in component-based software. They are all represented by sorts. Second, the language itself is independent of the software component models, or the programming languages used to implement the software entities. In the implementation of CASOCC language for testing components of a specific programming language and/or component model, pre-defined sorts can be introduced to represent the pre-defined classes/data types.

A specification in CASOCC consists of a number of modular units. Each unit specifies one software entity. This entity is represented by a sort, which is called the unit's main sort. Each unit defines the signature of the entity (i.e. the operators), a set of axioms that the operators must satisfy as well as a list of '*imported*' sorts, which represent the other software entities that it depends on. The list of imported sorts defines the importation relation on sorts and thus the dependence structure of the component-based software. The distinction between main sorts from imported sorts does not only decide which axioms are to be checked, but also plays a significant role in the generation of test cases. It is worth noting that importation is different from enrichment or extension operations of AS modules [24]. Instead, importation in CASOCC is equal to the *protected importation* operation on modules in CafeOBJ and OBJ3. The importation relation can also be supported indirectly by the composition mechanisms of CASL language [25].

The following EBNF formulas define the overall structure of a specification unit in CASOCC.

```
<Spec Unit> ::=
 Spec < Sort Name > Observable: <Boolean>
    [Import: <Import Sort List>]
    Operations: <Operation List>
    [Var:  <Variable Declaration List>]
    [Axioms: <List of axioms>]
 End
```

The *VAR* clause in a specification unit declares a list of universally quantified variables that occur in the axioms. Each variable declaration is in the form of *<variable identifier>*:*<sort name>*, where the sort name is either the main sort, an imported sort or a predefined sort. In the implementation of CASOCC language for testing Java Beans, the following pre-defined sorts are Java's primitive data types, which include *byte*, *short*, *int*, *long*, *float*, *double*, *char*, *String* and *Boolean*.

An axiom in CASOCC is a conditional equation in the following form.

```
<Axiom> ::= <Label> : <Equation>[, if  <Condition>]
<Label>::= <Number> | <Identifier>
<Equation> ::= <Term> = <Term>
<Condition> ::=  <Term of Boolean type>
    | <Equation> | <Term> <Relation Operator> <Term>
    | <Condition> <Logic Connective> <Condition>
```

A term can be formed from variables declared in the VAR clause and constants of predefined sorts by applying operators defined in the *Operator* clause and the operators of the predefined sorts and imported sorts. It is worth noting that, we use LOBAS's notation [18] for the representation of terms in OO style rather than the traditional functional style. Therefore, a term $f(x,y)$, i.e. an operator f applied to parameters x and y, is represented in the form of $x.f(y)$, if x is of the main sort. Details of the syntax of terms are omitted for the sake of space. The following is an example of CASOCC specification. It specifies a stack with bounded depth of 10 elements.

```
Spec Stack
  observable F;      import int, String;
  operations
    creator          create: String->Stack;
    constructor      push: Stack,int->Stack;
    transformer      pop: Stack->Stack;
    observer         getId: Stack->String; top: Stack->int;
                     height: Stack->int;
  vars  S: Stack; n: int; x: String;
  axioms
     1: create(x).getId() = x;
     2: findByPrimaryKey(x).getId() = x;
     3: create(x).height() = 0;
     4: S.push(n) = S; if S.height() = 10;
     5: S.pop() = S; if S.height() = 0;
     6: S.push(n).pop() = S; if S.height() < 10;
     7: S.push(n).top() = n; if S.height() < 10;
     8: S.push(n).height() = S.height()+1; if S.height() < 10;
     9: S.pop().height() = S.height()-1; if S.height() >0;
end
```

It is worth noting that, a specification in CASOCC is independent of the way that the entity is implemented. A unit can be implemented as a component, class or data type. A system may consist of entities of different kinds.

2.2 Behavioural semantics and observable sorts

The semantics of CASOCC language is the so called *behavioural semantics* [22, 37, 38]. Therefore, the sorts are classified into observable and non-observable. In CASOCC, the observability of a sort is explicitly specified by the *Observable* clause. To ensure the soundness of the semantics of an algebraic specification, if a sort is indicated to be observable, there must be an equality operator "==" provided by the implementation of the corresponding software entity. Formally, observable sorts must satisfy the following syntax and semantics constraints [22].

Definition 1. (*Observable sort*)
In an AS $<\Sigma, E>$, a sort s is an *observable sort* implies that there is an operation $\_ == \_ : s \times s \rightarrow Bool$ such that for all ground terms τ and τ' of sort s,

$$E \mid\!-(\,(\tau == \tau') = true) \quad \Leftrightarrow \quad E \mid\!-\,(\,\tau = \tau'\,).$$

An algebra A (i.e. a software entity) is a correct implementation of an observable sort s if for all ground terms τ and τ' of sort s,

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

Note that pre-defined sorts of Java primitive classes and data types are observable.

2.3 Support to automatic testing

As discussed in [22], the information about the sort observability plays a significant role in the automated algebraic testing. To further support automated testing, CASOCC requires operators divided into four types in their declarations in the *Operator* clause. These types of operators are given below.

– *Creators* create and initialise instances of the software entity. They must have no parameters of the main sort, but result in the main sort.
– *Constructors* construct the data structure by adding more elements to the data. A constructor must have a parameter of the main sort and results in the main sort. They may occur in normal forms.
– *Transformers* manipulate the data structure without adding more data. Similar to constructors, a transformer must have the main sort as its parameter and results in the main sort. However, they cannot occur in any normal forms.
– *Observers* enable the internal states of the software entity to be observed from the outside. They must have a parameter of the main sort but result in an imported sort.

To enable automated testing of software components, we require the formal specification is well structured and matches the structure of program. The following formally defines the notion of well-structured. These properties ensure that the test oracles based on observation contexts are valid in behavioural semantics [22].

Let $\mathcal{U}$ be a set of specification units in CASOCC and S be a set of sorts. For each sort $s \in S$, there is a unit $U_s \in \mathcal{U}$ that specifies the software entity corresponding to sort s. Let $\prec$ be the importation relation on S.

Definition 2. (Well founded specifications)
A sort $s \in S$ is *well founded* if s is observable, or for all s' in the import list of U_s, s' is an observable sort, or s' is well founded. A specification $\mathcal{U}$ is *well founded* if and only if the importation relation $\prec$ is a pre-order on the set S of sorts, and all sorts $s \in \Sigma$ are well founded. □

Definition 3. (Well structured specifications)
A specification $\mathcal{U}$ in CASOCC is *well structured* if it satisfies the following conditions.
(1) It is well founded;
(2) For every user defined unit $U_s \in \mathcal{U}$,
  (a) there is at least one observer in U_s;
  (b) for every axiom E in U_s, if the condition contains an equation $\tau = \tau'$, we must have $s' \prec s$, where s' is the sort of terms τ and τ'. □

A practice implication of the well founded and well-structured properties is that for all sorts there are finite lengths of observable contexts. Thus, constructing test oracles based on observable context is feasible.

3. Generation of Checkable Test Cases

This section first reviews the notions of observation contexts and checkable test cases. Then, we present the algorithm of test case generation.

3.1 Observation context

The notion of observation context can be formally defined as follows [17, 20, 22].

Definition 1. (Observation context)
A context 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, written as $C[t]$, is the term obtained by substituting t into the special variable □. An *observation context oc* of sort c is a context of sort c and the sort of the term oc is $s \prec c$. To be consistent on notations, we write $\_.oc$: $c \rightarrow s$ to denote an observation context oc. An observation context is *primitive* if s is an observable sort. In such cases, we also say that the observation context is *observable* and call the context *observable context* for short. □

The general form of an observation context oc is:

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

where f_1, ..., f_k are transformers of sort s_c, obs is an observer of sort c, and $f_1(...)$, ..., $f_k(...)$ are ground terms. A sequence of observation contexts oc_1, oc_2, …, oc_n, where $\_.oc_1$: $c \rightarrow s_1$, $\_.oc_i$: $s_{i-1} \rightarrow s_i$, $i = 2,\ldots,n$, can be composed into an observation context $\_.oc_1.oc_2. \ldots.oc_n$. In [20], such compositions of observation contexts are called *observation context sequences*. In this paper, we do not distinguish them.

A primitive observation context (i.e. an observable context) produces a value in an observable sort. For example, consider the specification of Stack given in the previous section. The following are observation contexts. Because the predefined sort Integer is observable, these observation contexts are primitive.

_.top(), _.pop().top(), _.pop().pop().top(),
_.height(), _.pop().height (), _.pop().pop().height().

There are usually an infinite number of different observation contexts for a given AS. We define the complexity of an observation context $\_.f_1(...).f_2(...)....f_k(...).obs(...)$ as the number k of transformers. For example, the complexity of observation context _.top() is 0, and the complexity of _.pop().pop().height() is 2. Given an upper bound k on complexity, the set of all observation contexts with complexity less than or equal to k can be mechanically generated from the signatures of the sorts.

3.2 Checkable test cases

The basic idea of algebraic testing is to use algebraic specification to generate two ground terms that are supposed to be equal according to the axioms. Each term can be interpreted as a sequence of procedure/method calls. The results of the sequences are then checked for their equivalence. If not, errors are detected.

However, a sort may represent a structured data, a class even a component. The equivalence between the results is not always directly checkable. For example, in the Stack example, Create('st').Push(1).Pop and Create('st') should be equivalent, because both result in an empty stack called '*st*'. However, stacks are structured data. They cannot be directly compared for equivalence.

One approach to this problem is to generate test cases regardless whether the equivalence of the results can be checked directly or not; see e.g. [17, 20]. If the results cannot be checked directly, a set of observation contexts are applied to both results to reduce the equivalence problem into a set of sub-problems of equivalency, which could be further reduced if necessary. For example, to test the equivalence between terms Create('st') and Create('st').Push(1).Pop, the observer height can be applied to both to obtain two integer values, which can be compared directly. However, this approach does not work well for component testing. An alternative approach is to generate test cases that are observable, i.e. the equality of the terms can be observed; see, for example, [26, 27] for theoretical study of the approach.

Existing techniques for class testing will generate two instances of a class for each test case; one represents the result of one sequence of method calls. This technique is not applicable to components because a component can only have one instance [2]. In almost all component models, such as in EJB and CCM (CORBA Component Model), the result of the first sequence of method calls cannot always be copied and saved for comparison with the second result. Therefore, in addition to requiring the terms in a test case are ground, we also require the results to be recordable and comparable, thus the notion of checkable test cases.

In general, a test case is a triple $<T_1, T_2, C>$, where T_1 and T_2 are ground terms and C is an optional condition, which is a ground term of *Boolean* sort. It means that values of T_1 and T_2 should be equivalent if C evaluates to *True*. For the sake of readability, in the sequel we write a test case in the form of $T_1=T_2$, [if C].

Definition 2. (Checkable test cases)
A test case $T_1=T_2$, [if C] is *directly checkable* (or simply *checkable*), if and only if
a. the sort of terms T_1 and T_2 is observable, and
b. the sort of equations in C is observable, if any. □

For example, in the following, test case (a) is not checkable, but test case (b) is checkable.

Create('st').Push(1).Pop = Create('st') (a)
Create('st').Push(1).Pop.Height = Create('st').Height (b)

3.3 Test case generation

In addition to the checkability problem, there is another problem for the generation of component test cases.

As discussed above, existing test case generation methods are essentially to substitute ground terms into variables of two terms that are supposed to be equivalent according to the axioms, such as the two sides of an axiom, or one is the normal form of the other [18]. However, there are some subtle differences in what are substituted into the variables in different techniques. DAISTS substitutes user-defined terms [16]. In [17], all ground terms of certain complexity are used. TACCLE

[20] only uses ground terms in normal forms. A problem with these approaches is that when operators have parameters of predefined data types, such as integers, using ground terms is not effective and practical. For example, in the form of normal form, the integer value 3 is represented as *succ*(*succ*(*succ*(0))). An integer value 2000 would be impossible to be used in a test case. Chen *et al.*'s solution to this problem [20, 21] is to apply white-box testing techniques to select values that cover all paths in the software under test. Unfortunately, this is not applicable to component testing because the source code is not always available. Therefore, we combine random testing with algebraic testing by selecting the values for variables of predefined data types at random. The following is the test generation algorithm.

Algorithm 1.

Input:
  Spec s: CASOCC specification unit of the main sort;
  Sigs $s_1, s_2, \ldots, s_k$: The signature of imported sorts;
  TC: A subset of axioms in s (* the axioms to be tested *);
  vc: Integer (*complexity upper bound of variables*);
  oc: Integer (*complexity upper bound of observation contexts*) ;
  rc: Integer (* the number of random values to be assigned to variables of primitive sorts*)
Output: ts: The set of test cases;
Begin
(* Step 1: Initialisation *)
  pv:= the set of variables in spec s of observable sorts.
  sv:= the set of variables in spec s of non-observable sorts.
(* Step 2: generate normal form terms for non-primitive variables*)
  for each variable $v \in sv$ **do**
    {$T(v)$:= *NormalForms*(v:s_v, vc);
      for each variable v' in $T(v)$ **do**
        if v' is of observable sort
        then add v' to pv **else** add v' to sv; **}**;
(* Step 3: generate random values for primitive variables *)
  for each $v \in pv$ **do** Generate a set $RV(v)$ of rc random values;
(* Step 4: Substitute normal forms into axioms *)
  for each $tc \in TC$ **do**
    {**for** each variable $v \in sv$ that occurs in tc **do**
      for each term $gt \in T(v)$ **do** TC:= $TC + tc[v/gt]$;
     Remove tc from TC;}
(* Step 5: Substitute random values into test cases *)
  for each $tc \in TC$ **do**
  {**for** each variable $v \in pv$ that occurs in tc **do**
    for each $u \in RV(v)$ **do** TC:= $TC+tc[v/u]$;
    Remove tc from TC;}
(* Step 6: Compose test case with observation context *)
  for each tc=<t_1=t_2; if c> $\in$ TC **do**
  { TCO := $\varnothing$;
    if t_1 and t_2 are not primitive
    then {OC:= *PObsContexts*($t1$:s, oc);
      for each obc **in** OC **do** TCO:= TCO+<$obc.t_1$=$obc.t_2$;if c>;
      if c is not primitive **then** POC :=*PObsContexts*(c:s, oc);
      for each tc=<t_1=t_2; if c>$\in TCO$ **do**
        TCO:=TCO+<t_1=t_2; if $\wedge${$POC.c$}>
    if $TCO \neq \varnothing$ **then** {TC:=$TC \cup TCO$; Remove tc from TC;} } **}**;
(* Step 7: output test set *)
  ts := TC;
End □

In the above algorithm, *NormalForms*(v:s_v, vc) is the set of normal forms of the sort s_v of v with complexity from 0 to vc. It is generated from the signatures of the creators and constructors of the sorts. *PObsContexts*(t_1:s, oc) is the set of primitive observation contexts of sort s with complexity from *0* to oc. It can be generated from the signatures of the transformers and observers of the sorts. The term $tc[v/gt]$ is obtained by systematically replacing v with term gt in tc. The term $\wedge\{POC.c\} \cong p_1.c \wedge p_2.c \wedge \ldots \wedge p_k.c$, if $POC=\{p_1, p_2, \ldots, p_k\}$.

The algorithm has the following properties. Their proofs are omitted for the sake of space.

Theorem 1. The test case generation algorithm will always terminate if the specification is well founded. □

Theorem 2. The test cases generated are checkable, i.e. for all test cases <t_1=t_2; if c> generated by the algorithm, t_1, t_2 and c are of primitive or observable sorts. □

The following theorem about the correctness of the algorithm can be derived from the theorems proven in [22]. Here, the software under test is assumed to be deterministic.

Theorem 3. The test cases are valid. That is, if the specification is well-structured and the observable sorts satisfy the constraints in Definition 1, we have the following properties.

(a) The program correctly implements the specification with respect to the behavioural semantics of algebraic specifications implies that the evaluation of t_1 and t_2 using the program give equivalent results provided that c is evaluated to be *true*.

(b) If the evaluation of t_1 and t_2 gives non-equivalent values in an implementation when c is evaluated to *true*, then there are faults in the program. □

The following gives some examples of the test cases for the Stack example generated by the CASCAT tool.

```
create(String:[gfn2785]).height() = int:[0];
create(String:[Rm8]).push(int:[961467407]).pop().top()
   = create(String:[Rm8]).top();
     if create(String:[Rm8]).height()<int:[10];
create(String:[Rm8]).push(int:[961467407]).pop().height()
   = create(String:[Rm8]).height();
     if create(String:[Rm8]).height()<int:[10];
```

4. Testing Tool CASCAT

Figure 1 shows the structure of prototype testing tool CASCAT (Common AS-based Component Automatic Testing) for testing Enterprise Java Beans (EJB) on the JBoss platform. It contains four main parts. *Specification Parser* parses the AS in CASOCC, and checks the well-formedness of the specification and the type correctness of equations in the axioms. *Test Case Generator* takes two parameters from the user and generates a set of test cases. The parameters are the upper bounds of the complexities of the observation contexts and the values substituted into the variables. *Test Driver* executes the component on each test case and records the test results. *Test Result Evaluator* checks the correctness of the results and reports to the user.

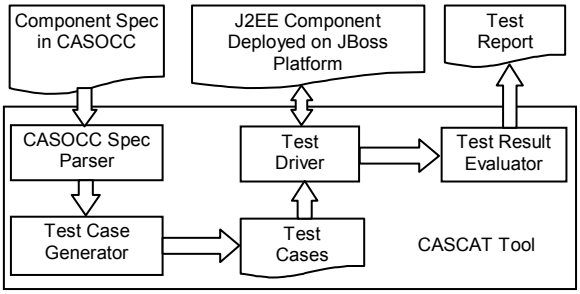

Figure 1. Overall Structure of CASCAT Tool

The inputs to the automated test process are a specification of the component, the component location deployed to JBoss platform, the location of the JBoss server, and the complexities of the observation contexts and the ground terms to be substituted into variables in the axioms. CASCAT also allows the user to select a subset of axioms for testing, thus to focus on a subset of functions and properties of the component. In such cases, test cases are generated from these selected axioms only. Figure 2 shows the interface of the tool.

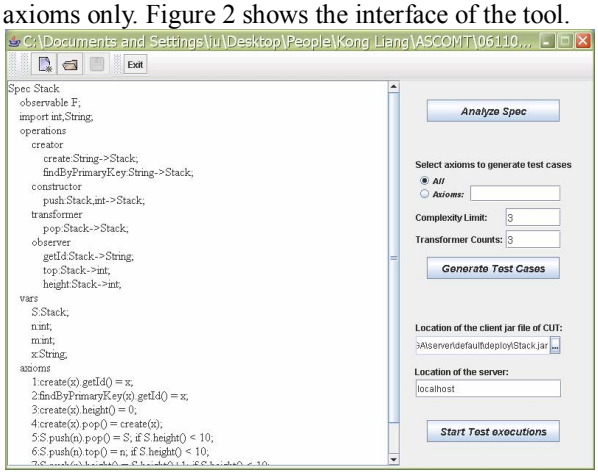

Figure 2. Interface of The CASCAT Tool

5. Evaluation of effectiveness

To evaluate the effectiveness of the testing method, we have carried out an experiment using the prototype tool. This section reports the experiment results.

5.1 The experiment process

The main goal of the experiment is to find out the fault detecting ability of the testing method. The experiment consists of the following activities.

- *Selection of subject components*. A number of software components from well established public sources were selected.
- *Development of formal specification*. For each subject component, a formal specification in CASOCC is developed based on the document and source code.
- *Test case generation*. A number of test sets are generated by the CASCAT tool from the specification.
- *Validation of formal specification*. The subject component is checked against its formal specification by executing the components on the test cases using the CASCAT tool. If any axiom not satisfied or the component terminates abnormally on a test case, the specification is considered as containing errors and revised. Then, the test cases are re-generated. The iteration continues until all axioms are satisfied.
- *Fault injection*. The mutation operators were applied to the source codes to insert the faults into the components. MuJava [28] is used to generate mutants as the faulty components.
- *Eliminate equivalent mutants*. Each mutant is manually examined to determine if it is equivalent to the original. Equivalent ones are removed.
- *Test execution*. Each test set is applied to each faulty component using the CASCAT tool. A component is classified as fault detected if at least on of the axioms of the component is violated or the execution is terminated abnormally. Note that this differs from classifying whether a mutant is killed according to whether produced the same output as the original.

5.2 The subjects

The subjects used in the experiments were retrieved from sources available to the public, such as the official guide to JBoss [29], J2EE [30], textbooks for professionals [31, 32, 33] and research papers [34].

We selected ten subjects that consist of a total of 20 components. They were selected for two reasons. First, they represent correct uses of the component technology as they are from the developers of the EJB technology. Therefore, the results of the case study can represent the best practice in component development rather than ad hoc uses of the technology. Second, the examples are selected for variety, i.e. to cover a wide range of application domains, of various complexities and to cover all types of components. Some examples contain only one component, some consist of several components. These subjects are briefly described below.

- *Bank*: It is a session bean that keeps records on the changes of balances of bank accounts [30].
- *College*: It is for applications in the context of university information systems, which keeps records about the students and the courses that they take. It consists of two components: *Course* and *Student* [30].
- *Order*: It is an entity bean in the context of online shopping applications [30].
- *Sales*: It consists of components *SalesRep* and *Customer* for applications in business management [30].
- *Stock*: It is an entity bean in the context of the stock market information systems [32].
- *Warehouse*: It consists of two components for warehouse applications. *Storage* models storage spaces.

Widget models the widgets stored in the spaces [30].

- *Gangster*: It is for a crime watch web portal application. It consists of two components. The *gangster* bean manages information about gangsters. The *readahead* creates html files to display the information [30].
- *Product*: The *Product* bean manages the information about products including the unique product ID, the name, description and base price of the product [33].
- *SafeDriver*: It is a set of components in the portal solutions of safedrive.com. It consists of four components. The *RateTable* entity bean deals with the rate of driver assurance. The *Generalinfo* session bean facilitates the clients to call register function. The *RateQuote* session bean calculates the premium value. The *Register* entity bean registers user details [31].
- *Cart*: It is a session bean that represents a shopping cart in an online bookstore [30].
- *Count*: It is a component for counting the number of events in a period of time [33].
- *LinkedList*: This session bean implements the operations on linked lists. The implementation is from a Java textbook. Its algebraic specification is from [34].
- *Math*:This session bean implements the basic mathematical functions in *java.lang.Math* of JDK 1.5.0.6.

Table 1 shows the scales of the components. The column *#C* gives the number of classes in the component. The column *#M* gives the number of methods in the component's interface. The column *#L* gives the total number of lines in the source code including the comments. The table also shows where the component comes from in the column *Src*.

Table 1. Subject components used in the experiment

Subject/Component		#C	#M	#L	Src	Type
Single Component Subjects						
Bank		4	4	322	[28]	Stateful
Stock		3	5	250	[30]	BMP
Product		5	15	278	[31]	CMP
Cart		5	6	226	[28]	Stateful
Count		3	2	101	[31]	Stateful
LinkedList		4	8	155	[32]	Stateful
Math		3	7	142	[34]	Stateless
Multiple Components Subjects						
College	Course	3	5	367	[28]	BMP
	Student	3	5	330	[28]	BMP
Sales	Customer	3	7	350	[28]	BMP
	Order	5	7	1082	[28]	BMP
	SalesRep	3	5	375	[28]	BMP
Warehouse	StorageBin	3	5	360	[28]	BMP
	Widget	3	4	291	[28]	BMP
Gangster	Gangster	6	27	214	[27]	CMP
	ReadAhead	4	7	234	[27]	Stateless
SafeDriver	RateTable	3	16	292	[29]	CMP
	Register*	4	19	293	[29]	CMP
	RateQuote	3	5	341	[29]	Stateful
	GeneralInfo*	4	2	290	[29]	Stateless
Average		3.7	8.6	315	--	--
Std Dev		0.9	6.3	196.4	--	--

There are three types of EJB. A session bean performs a task for a client and implements the business logic. An entity bean represents a business entity object that exists in persistent storage. A message-driven bean acts as a listener for the Java message service API and processes messages asynchronously [30]. An entity bean can be implemented in one of two ways: as a Bean Managed Persistence (BMP) entity bean, or as a Container Managed Persistence (CMP) entity bean. Session beans can also be classified into *stateful* session beans and *stateless* session beans. The components used in the experiment cover all these types of Java components except message-driven beans. The type of each component is also given in Table 1 in the column *Type*.

5.3 Development of algebraic specifications

In the experiment, we developed algebraic specifications based on the documents and source codes of the components, except the linked list.

Generally speaking, the development of an algebraic specification consists of two main tasks: (a) the description of the signature of the sorts and the classification of operators; (b) the determination of the axioms. The former involves the analysis of the interfaces of the components, the identification of the operations and the classification of operations into creators, constructors, transformers and observers. The first two steps are usually straightforward. The signature of the operations can be derived from the type definitions of the methods in Java classes. The classification of these methods requires understanding the semantics of the methods, but it is usually fairly easy in our case study.

The development of axioms was less difficult than we expected although it requires a deeper understanding of the semantics of the components. In our experiment, we noticed that there is a simple pattern of axioms for entity beans despite their differences in semantics.

In general, the operations of entity beans usually consist of a collection of setters and getters. For each setter $setX(v)$ for setting the value of attribute X to be v, we often can define an axiom in the following form to specify that after executing the setter operation, the value of X is indeed v.

$\forall s,v.\ (s.setX(v).getX = v)$,

where $getX$ is the getter method of attribute X. We can also to define axioms in the following form to specify that the setter does not change the values of other attributes, if there is no side-effect expected.

$\forall s,v.\ (s.setX(v).getY = s.getY)$,

where $getY$ is a getter method for another attribute Y.

For session beans, the semantics of a component represents the component's business logic in its application domain. Thus, it often requires more semantic analysis to gain a deeper understanding of their meanings. However, once understood the semantics of the operations, the specification of the meanings in CASOCC has not been a difficulty. Most predicates

specifying the pre/post-conditions of a component in design-by-contract approach can be straightforwardly translated into algebraic specifications. For example, the property of increase-by-one operation can be specified as follows in algebraic specification.

$\forall s.\ (s.Increase\ A.getA = s.getA+1)$.

Table 2 summarises the AS of the components. The number of creators is in column *#Crts*. The number of constructors and transformers is in column *#Con*/*Trans*; the number of observers in column *#Obs*; and the number of axioms in column *#Axioms*. The total amount of human effort spent on the development of specifications for each component is also given in the table in the column *Time*. The average time of writing the algebraic specification for a component is about 82 minutes.

5.4 Main findings

The main results of the experiments are shown in Table 3. Column *#M* gives the total number of mutants generated by the MuJava tool including method mutants and class mutants. Column *#NEM* is the total number of non-equivalent mutants. Column *#D* gives the number of mutants that the CASCAT tool detected being faulty, that is, the number of the mutants that do not satisfy the axioms. The column *Rate* gives fault detecting rate, i.e. the percentage of non-equivalent mutants that the CASCAT tool detected faults. The *total fault detecting rate* is calculated as the total number of detected mutants in all examples divided by the total number of non-equivalent mutants of all components. The *average fault detecting rate* is the division of the sum of the fault detecting rates of the components by the number of the components. The standard deviations are with respect to the averages. The experiment shows that according to both measures of total and average fault detecting rates, on average the fault detecting ability is around 80%. The experiment results are consistent with our preliminary experiments reported in [15].

Note that, *Register* and *GeneralInfo* components are abstract. MuJava does not generate mutants for abstract methods. But, CASCAT can still generate and execute test cases for them because abstract methods are implemented by the JBoss platform automatically when they are deployed. However, the fault detecting rates for such components do not truly reflect the effectiveness of the testing method. Therefore, they are excluded from statistical analysis of fault detecting ability.

One of the main findings is that the fault detecting ability is not sensitive to the scale of the subject under test. As shown in Figure 3, when the number of mutants increases, the average fault detecting rates stays at the same level. Statistical analysis of the correlation between the number of non-equivalent mutants and fault detecting rate shows that the Pearson product-moment correlation coefficient between the parameters is 0.20. Therefore, it is confident to state that fault detecting rate is independent of the complexity of the component.

Table 2. Summary of the algebraic specifications

Component	#Crts	#Con/Trans	#Obs	#Axioms	Time
Bank	1	1	2	6	60m
Cart	2	2	2	4	50m
Count	1	0	1	1	35m
Course	2	1	2	6	50m
Customer	2	2	3	4	60m
Gangster	2	5	20	19	90m
GeneralInfo	1	0	1	1	30m
LinkedList	1	2	5	16	100m
Math	1	1	5	9	60m
Order	2	0	5	5	60m
Product	2	3	10	14	70m
RateQuote	1	0	4	4	50m
RateTable	2	7	7	14	80m
ReadAhead	1	0	6	6	75m
Register	2	8	9	17	90m
SalesRep	2	1	2	3	40m
Stock	2	1	2	5	65m
StorageBin	3	0	2	4	40m
Student	2	1	2	4	40m
Widget	2	0	2	4	50m
Average	**1.75**	**1.75**	**4.60**	**6.85**	**82.25m**

Table 3. Results of experiments

Subject/Component		#M	#NEM	#D	Rate
Single component subjects					
Bank		42	37	23	62.2
Stock		23	22	13	59.1
Product		18	14	14	100.0
Cart		8	8	7	87.5
Count		13	9	8	88.9
LinkedList		33	33	32	97.0
Math		98	73	73	100.0
Multiple component subjects					
College	Course	24	24	14	58.3
	Student	25	25	15	60.0
Sales	Customer	40	40	27	67.5
	Order	57	53	34	64.2
	SalesRep	24	24	14	58.3
Warehouse	StorageBin	46	44	27	61.4
	Widget	34	28	20	71.4
Gangster	Gangster	9	8	8	100.0
	ReadAhead	35	33	31	93.9
SafeDriver	RateTable	48	36	36	100.0
	RateQuote	272	220	202	91.8
Total		849	731	598	81.8
Average		47.2	40.6	33.2	79.0
Std Dev		59.9	47.7	44.9	17.6

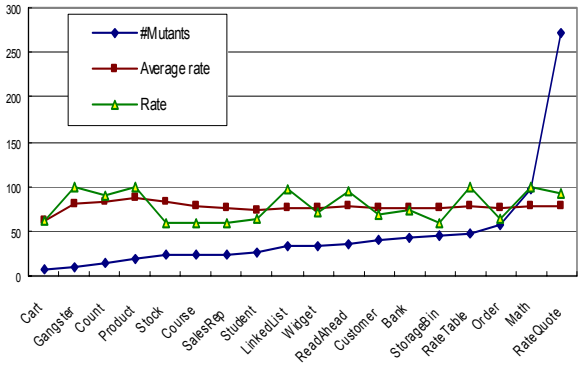

Figure 3. Complexity vs fault detecting rate

Figure 4 shows how the tests of single component subjects are compared with the tests of multiple component subjects. As one would expect, fault detecting ability seems decreasing when testing multiple component subjects. However, statistical analysis of the above statement was not conclusive, thus more research on this is necessary.

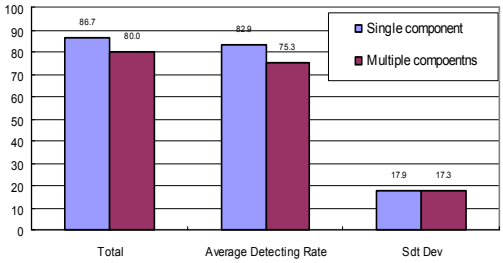

Figure 4. Single component vs multiple components

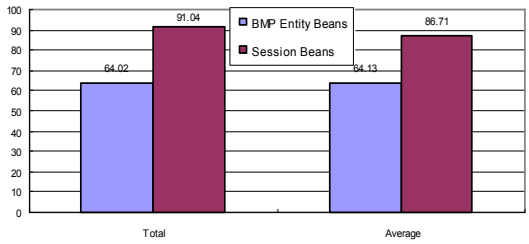

Figure 5. BMP Entity Beans vs Session Beans

A surprising finding is that the method consistently detects significantly more faults in session beans than in BMP entities beans despite that entity beans are usually much less complex than session beans, as shown in Figure 5. A statistical analysis using t-test shows that with probability less than 0.003 the above statement is false. One of the possible reasons is that the generated mutants better represent the type of errors that developers are likely to make in writing session beans than in writing entity beans. For example, the mutants represent commission errors are generated to setters and getters methods of entity beans. Specification-based testing methods are less capable of detecting them, but such faults are less likely to occur in practice.

6. Conclusion

This paper developed a technique of automated component testing based on algebraic specifications. A specification language CASOCC is designed to support testing software components. An algorithm is developed to generate checkable test cases. An automated prototype testing tool CASCAT is implemented for testing EJB components. The approach has the following advantages. First, AS are independent of the implementation details, thus suitable for software components. Second, as shown by the CASCAT testing tool, algebraic testing of components can achieve a very high degree of automation, which include test case generation, test driver construction and test result checking. Third, it does not require the availability and uses of the full set of axioms of all constituent and dependent entities. Thus, it is scalable and practically usable. Moreover, it allows software testers to focus on a subset of functions and properties of the component under test. Finally, the method can achieve a high fault detecting ability as shown by our experiment.

6.1 Related work

The most closely related work includes the research on algebraic testing and component testing.

In the area of algebraic testing, there are DAISTS in early 1980s [16] and Gaudel *et al.'s* work in late 1980s [17] on testing abstract data types, Frankl and Doong's LOBAS language and ASTOOT system [18], Hughes and Stotts' Daistish [19] for testing classes of OO programs, and more recently, Chen, *et al.*'s TACCLE [20, 21] for testing clusters of classes. Algebraic testing has not been applied to component testing before. Existing techniques of algebraic testing are not readily applicable to component testing as discussed in the paper. In [26, 27], the notion of checkable test cases (called observable test cases) is studied theoretically. Testing based on structured algebraic specifications in CASL has also been investigated theoretically by Machado *et al* [35, 36]. There are no implementations or empirical studies for testing components to our knowledge. Table 4 compares our technique with existing test tools that are based on algebraic specifications.

Table 4. Comparison with existing works

Work /System	Entities tested	Test case generation	Result checking
DAISTS [16]	ADT	Manually scripted ground terms substituted into axioms	Manual program
LOFT [17]	ADT	All ground terms up to certain complexity substituted to axioms; random values assigned to variables of predefined sorts.	Observation contexts
ASTOOT / LOBAS[18]	Class	Rewriting ground terms into their normal forms using the axioms	Manual program
Daistish [19]	Class	Manually scripted ground terms substituted into axioms	Manual program
TACCLE [20, 21]	Class, Cluster	Ground normal forms substituted in axioms + path coverage	Observation context
CASOCC/ CASCAT	ADT, Class, Component	Composition of observation contexts and axioms with ground normal forms substituted into non-primitive variables and random values for primitive variables	Direct checking since test cases are checkable

The speciality of testing component-based systems is that when a component is integrated into a system, its specification should not be effected by the context in which the component is used. This is exactly what the protected importation mechanism is as a specification composition mechanism. Other specification composition mechanisms such as unprotected importation, union, renaming and translation mechanisms in CASL that are useful for software architectural specifications seem

less useful in the context of component-based development and testing. Thus, the importation relation is used in CASOCC language.

Comparing with other specification-based component testing methods such as those based on design-by-contract [13] and finite state machines [14], our method has the advantage of high degree of automation. In comparison with self-testing techniques such as [12], specification-based testing methods do not introduce additional complexity into the code.

6.2 Future work

We are planning more experiments with software that contains multiple components. Our tool is implemented for testing EJB 2.0 component. It does not directly support message driven components defined in EJB 3.0. Therefore, the case study does not include message driven components. We are extending the implementation of the tool to enable direct testing of such components. We are also interested in extending the technique for testing other types of systems such as web services and concurrent systems. As stated in [37, 38], the theories of behavioural algebraic specifications can be applied to a wider range of software systems including concurrent and non-deterministic systems.

7. References

[1] Hopkins,J., Component primer, C.ACM 43(10), 27-30.

[2] Szyperski,C., Component Software: Beyond Object Oriented Programming, 2nd Edition, Addison Wesley, 2002.

[3] D'Souza, D. and Wills,A.C., Objects, Components and Frameworks with UML, Addison Wesley, 1999.

[4] Gao,J., Tsao, J., & Wu, Y., Testing and Quality Assurance for Component-Based Software, Artech House, 2003.

[5] Sparling,M., Lessons learned through six years of component-based development, C.ACM 43(10), 2000, 47-53.

[6] Crnkovic,I., Larsson,M., A case study: demands on component-based development, ICSE'2000, 22-30.

[7] Morisio, M., *et al*., Investigating and improving a COTS-based software development, ICSE'00, 32-41.

[8] Beydeda S. & Gruhn, V. (eds), Testing Commercial-Off-The-Shelf Components and Systems, Springer, 2005.

[9] Zhu, H. and He, X., A Methodology of Component Integration Testing, in [8], Springer, 2005, 239-269.

[10] Beydeda, S. & Gruhn, V., State of art in testing components, QSIC'03, 146-153.

[11] Zhu, H. and He, X., An Observational Theory of Integration Testing for Component-Based Software Development, COMPSAC'01, 2001.

[12] Beydeda, S., Self-Metamorphic-Testing Components, Proc. COMPSAC'06, 2006, 265-272

[13] Briand, L. C., Labiche, Y., Sówka, M. M., Automated, Contract-based User Testing of Commercial-Off-The-Shelf Components, ICSE'06, 92-101.

[14] Gallagher, L. and Offutt, J., Automatically testing interacting software components, AST'06, 57-63.

[15] Kong,L., Zhu,H. & Zhou,B., Automated Testing EJB Components Based on Algebraic Specifications, COMPSAC'07, Vol. 2, 717-722.

[16] Gonnon, J., McMullin, P. and Hamlet, R., Data-Abstraction Implementation, Specification and Testing, ACM TOPLAS 3(3), 1981, 211-223.

[17] Bernot, G., Gaudel, M. C., and Marre, B., Software testing based on formal specifications: a theory and a tool, Software Engineering Journal, Nov. 1991, 387- 405.

[18]Doong,K.&Frankl,P., The ASTOOT approach to testing object-oriented programs,ACM TSEM3(2),1994, 101-130

[19] Hughes, M. and Stotts, D., Daistish: systematic algebraic testing for OO programs in the presence of side-effects. ISSTA'96, 53-61.

[20] Chen,H.Y., Tse,T.H. & Chen,T.Y., TACCLE: a methodology for object-oriented software testing at the class and cluster levels, ACM TSEM 10(1), 2001, 56-109.

[21] Chen, H.Y., et al., In black and white: an integrated approach to class-level testing of object-oriented programs, ACM TSEM 7(3), 1998, 250-295.

[22] Zhu, H., A Note on Test Oracles and Semantics of Algebraic Specifications, QSIC'03, 91-99.

[23] Goguen, J. A., et al. Initial Algebra Semantics and Continuous Algebras, J. ACM 24(1), 1977, 68 – 95.

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

[25] Bidoit,M., Sannella,D. & Tarlecki,A., Architectural Specifications in CASL, Formal Aspects of Computing 13, 2002, 252-273.

[26] Le Gall, P. and Arnould, A., Formal specification and test: correctness and oracle, 11th WADT/9th COMPASS Workshop, LNCS 1130, Springer 1996, 342-358

[27] Machado, P, On oracles for interpreting test results against algebraic specifications, AMAST'98, LNCS 1548, Springer, 1998, 502-518.

[28] Ma,Y-S., Offutt,J. & Kwon,Y-R., MuJava: An Automated Class Mutation System, STVR,15(2):97-133, 2005.

[29] Fleury, M., Stark, S., and Richards, N., JBoss 4.0 -- The official Guide, Pearson Education, 2005.

[30] Bodoff,S.,et al.,The J2EE Tutorial,2nd Edt.,Pearson 2004.

[31] Kumar, B. V., Sangeetha, S., Subrahmanya, S. V., J2EE Architecture, Apress 2003

[32] Mukhar,K.,Zelenak,C.,Weaver,J. L.Crume,J., Beginning Java EE 5 From Novice to Professional, Apress, 2005.

[33] Srignaesh, R. P., Brose, G., Silverman, M., Mastering Enterprise Java Beans 3.0, Wiley Publishing, Inc., 2006.

[34] Henkel, J., Discovering and Debugging Algebraic Specifications for Java Classes, PhD thesis, University of Colorado at Boulder, USA, 2004.

[35] Machado,P.,Testing from Structured Algebraic Specifications,AMAST2000,LNCS1816,Springer,2000,529-544

[36] Machado,P. and Sannella,D., Unit Testing for CASL Architectural Specifications, Mathematical Foundations of Computer Science,LNCS 2420,Springer, 2002,506-518

[37] Goguen,J. and Malcolm,G., A hidden agenda, Theoretical Computer Science 245(1), 2000, 55-101.

[38] Sannellla,D., Tarlecki,A., On observational equivalence and algebraic specification, J. Computer & Systems Science 34, 1987, 150-178.