Automated Testing EJB Components Based on Algebraic Specifications

Liang Kong(2), Hong Zhu(1) and Bin Zhou(2)

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

(2) Department of Computer Science, National University of Defence Technology
Changsha, China, Email: liangkong@gmail.com

Abstract

Algebraic testing is an automated software testing method based on algebraic formal specifications. It has the advantages of highly automated testing process and independence of the software's implementation details. This paper applies the method to software components. An automated testing tool called CASCAT for Java components is presented. A case study of the tool shows the high fault detecting ability.

1. Introduction

In recent years, software component technology has emerged as a key element in the development of large and complicated systems [1, 2, 3]. Ensuring the quality of software components and their correct integration is a critical problem in component-based software development (CBSD). Industrial practices in CBSD have shown a clear shift of focus from design and coding to requirements analysis, integration and testing [4, 5, 6, 7].

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

In addition to manual component testing methods based on informal documentation, existing research has explored uses of formal specifications of software components such as design-by-contracts with pre/post-conditions [12] and state transition diagrams [13]. These methods are capable of automatic generation of adequate test cases with respect to certain adequacy criteria. However, they offer little support to checking the correctness of test results automatically, which is particularly important as behaviour observation is difficult in component testing.

In this paper, we explore a method of testing software components based on algebraic formal specifications. We report a prototype automated testing tool called CASCAT for testing Java Enterprise Beans and a preliminary case study of its fault detecting ability. The main advantage of the method is its full automation of testing process, including test case generation, test driver construction, and test result checking. Another distinctive feature of the testing method is that it allows testing to focus on a specific property of the component or a specific part of the functionality provided by component. This is particularly important because components are often designed for a broad applicability, but it is often that only a subset of a component's functions is actually used [12].

The remainder of the paper is organised as follows. Section 2 introduces the basic ideas of the testing method. Section 3 briefly reviews existing related work. Section 4 discusses the special issues in the application of methods to users' testing of components. A specification language CASOCC is presented. Section 5 presents the automated testing CASCAT tool for EJB components. Section 6 reports a case study of the testing tool. Section 7 concludes the paper with a discussion of further work.

2. Related works

Algebraic specification (AS) emerged in the 1970s as a formal method for the specification of abstract data types [14, 15]. It has now developed into a formal method for a wide range of software [16]. 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* on the sorts, and E is a set of axioms in the form of conditional equations that define the semantics of the operators.

In the application of AS method to the formal specification of data types, a sort represents a data type; operators represent the operations on the data type and constants, which are 0-ary operators. In application of the method to OO software, a sort represents a class and the operators represent methods of the class. The attributes are assumed to be accessed through get and set methods. In this paper, to apply the specification method to software components, we use sort to represent software components and the operators to the operations provided by the components.

The uses of AS in software testing can be back dated to early 1980s [17]. Since then, significant progresses have been made in the techniques that support automated software testing using AS, which is called *algebraic testing* in the sequel. Its basic idea is based on the fact that each ground term (i.e. a term without any free variables) of a given signature has two interpretations in the context of software testing. First, a ground term represents a sequence of calls to the operations. Second, it also represents a value, i.e. the result of the execution. Therefore, checking whether an equation is satisfied by an implementation on a test case meant to execute the operation sequences of the terms on both sides of the equation with variables substituted by the test data represented in the form of ground terms and then to compare the results. If the results are equivalent, the program is correct on the test case; otherwise, it has errors.

In early 1980s, Gonnon, McMullin and Hamlet developed a compiler based system called DAISTS to use AS in testing abstract data types implemented in procedural programming languages [17]. In DAISTS, each axiom was tested on manually scripted test cases and the results from both sides of the axiom were compared by calling a manually programmed TypeEquality function according to the data type of the result.

In late 1980s, Gaudel *et al.* further developed the theory and method of algebraic testing [18]. They used observation contexts [19, 20] to enable automatic comparisons of structured values without manually programming equality functions.

Algebraic testing received much attention since 1990s in the context of class testing of object-oriented (OO) programs. Frankl and Doong [21] and Hughes and Stotts [22] studied the effectiveness of testing OO programs based on AS. Hughes and Stotts [22] adapted DAISTS and developed a tool called Daistish for testing programs in Eiffel and C++ using the specification language of DAISTS. Frankl and Doong adopted a notation to represent AS in a form that is suitable for OO systems and developed an AS language called LOBAS and a tool called ASTOOT. They proposed an extension of the method to include negative test cases, which consists of two terms that are supposed to generate non-equivalent results. In [23, 24], Chen, Tse and Chen further developed the theory and method of automatic derivation of test oracles based on observation contexts. Through a counter-example, they raised a theoretical question about the validity of test oracles based on observation contexts. This question is answered by Zhu in [25]. It was proved that the test oracle is valid in the final semantics and behavioural semantics of AS.

A common weakness of existing algebraic testing techniques is that software is tested in a 'big bang' approach, i.e. all classes of a system is tested all together without employing any incremental integration strategy. This seriously limited the practical usability of the testing method. In [25], Zhu addressed this problem by proposing an approach to the organisation of AS to match the structures of software systems. Equations in an AS are divided into groups that each group represents a class in object-oriented system. A partial ordering between sorts is introduced to represent the importation relationship between classes. This partial ordering generalises the notion of observable sorts and supports incremental integration testing. This approach is implemented in this paper and extended by using sorts to represent classes, data types, as well as components.

3. Testing Method

This section gives the details of the testing method.

3.1 Specification Language CASOCC

A software component, as defined by Szyperski in [2], 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.' 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 creates a COTS component market. However, industrial standards of component models rarely specify the semantics of the functionality provided and required by a component. In this subsection, we discuss how to support automated testing by using AS of software components. We will present a simple specification language for this purpose.

Our AS language is called CASOCC. It stands for Common AS of Components and Classes. It 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. It is also independent of the software component models, or the programming languages used to implement the software entities. A specification in CASOCC consists of a number of modular units. Each unit specifies one sort by defining the operators on the sort and a set of axioms that these operators must satisfy. The components or classes that a component or a class depends on are specified as imported sorts. The importation relationship between the sorts is required to be acyclic. The following defines the overall structure of a unit in CASOCC.

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

In a CASOCC unit, the *Sort Name* is the sort specified by this specification unit, which is called the *main sort* of the unit. The *Import* clause declares a list of sorts that the main sort depends on. These imported sorts could be the data types or classes of the parameters of the operators defined in the unit. It defines the importation relation on sorts and thus the 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 derivation of test oracles. It is worth noting that importation relation is different from classic enrichment or extension operations of AS modules [16]. In stead, importation in CASOCC is equal to the *protected importation* operation on modules in CafeOBJ and OBJ3.

As discussed above, the observability of a sort plays a significant role in the automated algebraic testing. In CASOCC specifications, each sort is explicitly specified as either observable or not by using the *Observable* clause. A software entity is observable, if there is an equality operator "==" defined on the entity. Formally, observable sorts are defined as follows [25].

Definition 1. (*Observable sort*)

In an AS $<\Sigma, E>$, 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,

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

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

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

The *VAR* clause 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 an imported sort or a predefined sort of the CASOCC language, which include Java's primitive data types such as *byte*, *short*, *int*, *long*, *float*, *double*, *char*, *String* and *boolean*. They are observable.

To further support automated generation of observation contexts, CASOCC required operators divided into four types in their declarations in the *Operator* clause. These types of operators are given below.

– *Creators* create instances of the software entity, such as an object of a class, and/or initialise the entity, such as the initialisation of a software component. 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. For example, the push operator on stacks is a constructor. A constructor must have a parameter of the main sort and results in the main sort. It may occur in the normal forms if the axioms are used as term rewriting rules.
– *Transformers* manipulate the data structure without adding more data. For example, the pop operation on stacks is a transformer. Similar to constructors, a transformer must have the main sort as its parameter and results in the main sort. However, it cannot occur in any normal forms.
– *Observers* enable the internal states or data in the software entity to be observed from the outside. For example, the top and the depth operator on stacks are observers. Observers must have a parameter of the main sort but result in an imported sort.

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 Connectives> <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 adopted LOBAS's notation for the representation of terms in the object oriented style rather than the traditional functional style [21]. 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 specification in CASOCC. 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, specifications in CASOCC are independent of the implementations. A unit can be implemented as a component, a class or a data type. A system may consist of units implemented in different types.

3.2 Observation Contexts

To enable automated testing of software components, we require the formal specification is well structured and matches the structure of program. The following will first formally define the notion of well-structuredness.

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

Definition 2. (Well founded specification)
A sort $s \in S$ is *well founded*, if 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 $s \in \Sigma$ is well founded. □

Definition 3. (Well structured specification)
A specification $\mathcal{U}$ in CASOC 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 declared in U_s;
(b) for every axioms of conditional equations 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'. □

The notion of observation context can be formally defined as follows.

Definition 4. (Observable context)
A context $C[\ldots]$ of a sort c is a term C with one occurrence of a special variable $\square$ of sort c. The value of a term t of sort c in the context of $C[\ldots]$, written as $C[t]$, is the term obtained by substituting t into the special variable $\square$. An *observation context oc* of sort c is a context of sort c and the sort of the term $oc[\ldots]$ is $s \prec c$. To be consistent with our notation for operators, we write $\_.oc$: $c \rightarrow s$ to denote such an observation context $oc[\ ]$.

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

The general form of an observable 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 primitive observation 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().
```

It is worth noting that 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 in the context. For example, the complexity of observation context `_.top()` is 0, and the complexity of `_.pop().pop().height()` is 2.

3.3 Generation of Test Cases and Oracles

A simple but effective test case generation method used by many researchers is to substitute ground terms into variables in the axioms. The result of substitution is a pair of ground terms that should be equivalent according to the specification plus an optional condition that contains no variables. When the condition is evaluated to be true, then two sequences of method calls on the two sides of an equation are invoked separately to generate two results and further comparisons of the results are made to determine if the implementation is correct.

However, there are some subtle differences in what are substituted into the variables in the literature. In DAISTS, user-defined terms are used [17]. In [18], all ground terms of certain complexity are used. In TACCLE system [23], only ground terms in normal forms are used. A common problem with these approaches is that when operators have parameters of predefined data types, such as integers, there may be a large number of ground terms even in the normal form. Chen *et al.*'s solution is to apply white-box testing methods to cover all paths in the software under test. Unfortunately, this solution is not applicable to testing software components 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.

Another feature of software components that differs components from classes is that a component can only have one instance while a class may have many instances [2]. This has caused a technical problem in the application of algebraic testing to components. A typical implementation of testing tools for class testing will generate two instances of a class for each test case based on one axiom; one represents the result of a sequence of method calls corresponding to the left-hand-side of an equation, and the other represents the result of the right-hand-side. This technique is not applicable to components because the component can only have one instance. The result of the first sequence of method calls cannot always be recorded in almost all component models, such as in Enterprise Java Beans (EJB) and CCM (CORBA Component Model). A solution to this problem is that, instead of simply substituting the ground terms into variables in the axioms, primitive observation contexts are also applied to both left hand and right hand sides so that the results in predefined data types become recordable and comparable. The following gives some examples of the test cases for the Stack example.

```
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];
```

This approach also differs from existing works, which separates test cases from test oracles.

4. Testing Tool CASCAT

A prototype testing tool called CASCAT (Common AS-based Component Automatic Testing) has been designed and implemented in Java for testing Java Enterprise Beans running on the JBoss platform. As shown in Figure 1, it contains four main components.

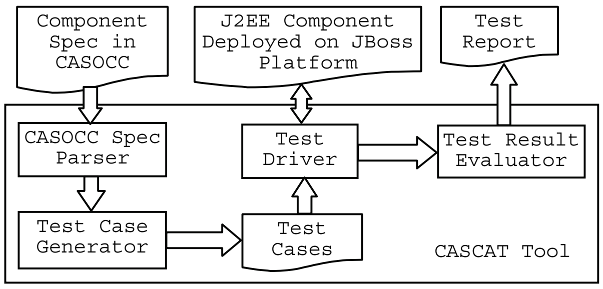
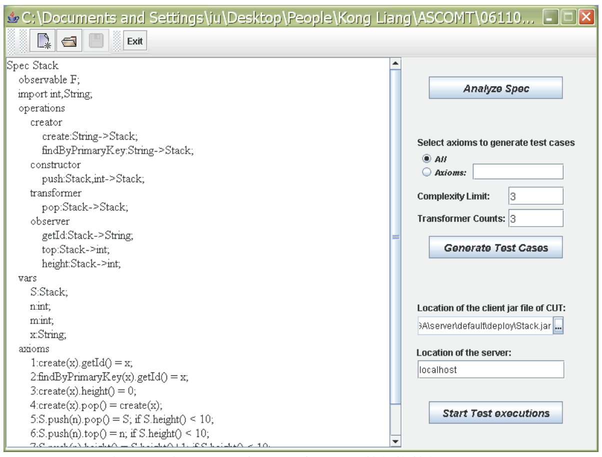


Figure 1. Overall Structure of CASCAT Tool

– *Specification Parser*: It 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*: It 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*: It executes the component on each test case and records the test results.
– *Test Result evaluator*: It checks the correctness of the results of the test executions and reports to the user.

The inputs to the automated test process are a specification of the component, the file location of the component 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. Figure 2 shows the interface of the tool.


C:\Documents and Settings\iu\Desktop\People\Kong Liang\ASCOMT\06110...
Exit
Analyze Spec
Select axioms to generate test cases
All
Axioms:
Generate Test Cases
Start Test executions

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

Complexity Limit: 3
Transformer Counts: 3
Location of the client jar file of CUT: ›A\server\default\deploy\Stack.jar
Location of the server: localhost

Figure 2. Interface of The CASCAT Tool

CASCAT allows the user to select a subset of axioms for testing, thus to focus testing efforts on a subset of functions and properties of the component. In such cases, test cases are generated from these selected axioms only.

5. Experiments

To evaluate the effectiveness of the testing method, we have carried out a preliminary case study using error seeding and mutation testing.

In the error seeding experiment, we manually inserted 37 faults into the Stack component and tested faulty components one by one against the axioms given in Section 3. Table 1, shows the distribution of the fault according to the classification proposed in [26]. All faults were detected. In the experiment, most injected faults are detected due to equations are not satisfied, while three path selection faults caused abnormal executions of the component on test cases. One path selection faults caused the component to execute without termination on a test case. Two other faults raised runtime error on test cases.

In mutation analysis, the MuJava testing tool is applied to the source code of the component to generate mutants of the component. Each component is then tested by CASCAT. The mutation analysis also demonstrated that the testing method can have an overall mutation score of 89.8% (killed 89.8% non-equivalent mutants). It indicates a high fault detecting ability. As shown in

Table 2, the result is consistent with the error-seeding experiment.

Table 1. Results of Error Seeding Experiment

#Faults / Type of Faults		Inserted	Detected	
			Normal	Abnormal
Domain Faults	Path Omission	3	3 (100%)	
	Path Selection	18	15 (83.3%)	3 (16.7%)
Computation Faults	Wrong Variable	3	3 (100%)	
	Wrong Expression	2	2 (100%)	
	Stmt Omission	7	7 (100%)	
	Stmt Transition	4	4 (100%)	
Total		37	34 (91.9%)	3 (8.1%)

Table 2. Results of Mutation Analysis

Mutant Type	#Total Mutants	#Equiv Mutants	#Killed Mutants		Mutation Score
			Normal	Abnormal	
AODU	2	0	2	0	100%
AOIS	44	2	20	18	90.5%
AOIU	5	0	4	1	100%
AORB	12	0	11	1	100%
AORS	5	0	2	3	100%
COR	2	0	0	2	100%
LOI	16	1	7	8	100%
ROR	5	1	1	3	100%
JSD	1	0	0	0	0%
JSI	5	0	0	0	0%
JTD	2	0	2	0	100%
JTI	4	1	3	0	100%
Total	103	5	52	36	89.8%

One of the purposes of our experiment with the tool is to find out how the complexities of test cases are related to fault detecting ability. In the experiment, we tested the component by first using test cases of the lowest complexities and then increasing the complexity of the test cases gradually until all faults are detected. We found that the majority of faults can be detected by test cases of very low complexity as shown in Table 3. The only faults that require test cases of high complexity to detect are in the piece of code that deals with the bound of the stack. Table 3 also shows that the number of test cases increases as the complexity increases.

Table 3. Fault Detecting Ability vs Complexity

Test case Complexity	#Test Cases	Mutation Analysis		Error Seeding	
		#Mutants Detected	%Mutants Detected	#Faults Detected	%Faults Detected
1	45	53	60.2%	22	59.5%
2	87	30	34.2%	14	37.8%
3	129	4	4.5%	0	0%
4~10	--	0	0	0	0%
11	465	1	1.1%	1	2.7%

6. Conclusion

In this paper, we explored the application of algebraic testing method to software components. A specification language CASOCC is designed. An automated EJB component testing tool CASCAT is implemented. The approach has the following advantages. First, AS are independent of the implementation details, thus are suitable for formal specification of 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 automation and test oracle generation. Moreover, it allows software testers to focus on a subset of functions and properties of the component that they are used. Finally, the method can achieve a high fault detecting ability as shown by our preliminary experiment with testing a software component, which confirmed the experiments done by Doong and Frankl [21].

For further research, we are conducting more experiments with the testing method using the prototype tool and analysing the types of faults that cannot be easily detected. We believe that the theories of observational AS can be applied to a wider range of software systems such as concurrent systems, because concurrency and non-determinism can be treated naturally by behavioural theories [19].

7. References

[1] Hopkins, J., Component primer, C.ACM 43(10), Oct. 2000, pp27-30.

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

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

[4] Gao, J., Tsao, J. H.-S., and 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), Oct.2000, pp47-53.

[6] Crnkovic, I., Larsson, M., A case study: demands on component-based development, Proc. ICSE'2000, June, 2000, pp22-30.

[7] Morisio, M., Seaman, C. B., Parra, A. T., Basili, V. R., Kraft, S. E. and Condon, S. E., Investigating and improving a COTS-based software development, Proc. ICSE'2000, June 2000, pp32-41.

[8] Beydeda S. and Gruhn, V. (eds), Testing COTS Components and Systems, Springer, 2005.

[9] Beydeda, S. and Gruhn, V., State of art in testing components, Proc. of QSIC03, Oct. 2003, IEEE CS, pp146-153.

[10] Beydeda, S., The Self-Testing COTS Component STECC Method, PhD Thesis, Department of Computer Science, University of Leipzig, Germany, 2004.

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

[12] Briand, L. C., Labiche, Y., Sówka, M. M., Automated, Contract-based User Testing of Commercial-Off-The-Shelf Components, Proc. of ICSE'06, May, 2006, Shanghai, China, IEEE CS Press, pp.92-101.

[13]Gallagher, L. and Offutt, J., Automatically testing interacting software components, Proc. AST'06, ACM Press, May 2006, pp.57-63.

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

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

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

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

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

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

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

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

[22] Hughes, M. and Stotts, D., Daistish: systematic algebraic testing for OO programs in the presence of side-effects. Proc. ISSTA'96, ACM Press, January 1996, pp53-61.

[23] Chen, H. Y., Tse T. H. and Chen, T. Y., TACCLE: a methodology for object-oriented software testing at the class and cluster levels, ACM TSEM 10(1), Jan. 2001, pp56-109.

[24] Chen, H.Y., *et al.*, In black and white: an integrated approach to class-level testing of object-oriented programs, ACM TSEM 7(3), Jul. 1998, pp250-295.

[25] Zhu, H., A Note on Test Oracles and Semantics of Algebraic Specifications,Proc.QSIC'03,Dallas,USA,Nov.2003,pp91-99.

[26] Harrold, M. J, Offutt, J, A. and Tewary, K. (1997) An approach to fault modeling and fault seeding using the program dependence graph, J. of Systems & Software 3, pp.273-296.