Applying Interface-Contract Mutation in Regression Testing of Component-Based Software

Shan-Shan Hou[1,2], Lu Zhang[1,2], Tao Xie[3], Hong Mei[1,2], Jia-Su Sun[1,2]
[1]*Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China*
{houss, zhanglu, meih, sjs}@sei.pku.edu.cn
[2]*Key laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, 100871, China*
[3]*Department of Computer Science, North Carolina State University, Raleigh, NC 27695*
xie@csc.ncsu.edu

Abstract

Regression testing, which plays an important role in software maintenance, usually relies on test adequacy criteria to select and prioritize test cases. However, with the wide use and reuse of black-box components, such as reusable class libraries and COTS components, it is challenging to establish test adequacy criteria for testing software systems built on components whose source code is not available. Without source code or detailed documents, the misunderstanding between the system integrators and component providers has become a main factor of causing faults in component-based software. In this paper, we apply mutation on interface contracts, which can describe the rights and obligations between component users and providers, to simulate the faults that may occur in this way of software development. The mutation adequacy score for killing the mutants of interface contracts can serve as a test adequacy criterion. We performed an experimental study on three subject systems to evaluate the proposed approach together with four other existing criteria. The experimental results show that our adequacy criterion is helpful for both selecting good-quality test cases and scheduling test cases in an order of exposing faults quickly in regression testing of component-based software.

1. Introduction

In software maintenance, regression testing is frequently applied to ensure software quality. Due to limited budget or desired efficiency, software engineers usually select a subset of test cases from the original test suite [15] or prioritize existing test cases [30] in regression testing. Typically, both tasks use some test adequacy criteria.

With the emergence of black-box components, such as reusable class libraries, commercial off-the-shelf (COTS) components, and web services, whose source code is not available, integrating black-box components to build new software systems has become an important way of software development. However, existing test adequacy criteria may not be effective to identify faults caused by misunderstandings between component providers and users.

Let us consider the following piece of code, in which two black-box functions (i.e., *abs* and *log*) are used to define a function named *f*. Here, *abs* returns the absolute value of its input and *log* returns the logarithm of its input. Due to the misunderstanding of the *log* function, the author of function *f* thinks that *log* can take any non-negative value as input, while *log* actually can take only positive values as input. As a result, there is a bug: taking *0* as the input causes a failure in *f*.

```
double f (int n){
   int a=abs(n);
   return log(a);
}
```

Statement coverage and branch coverage are not good criteria for testing *f*, as any input for *f* could achieve one hundred percent statement coverage and branch coverage. In fact, if we treat *abs* and *log* only as black-box functions without considering that *abs* returns a non-negative value and *log* can take only positive values as input, the input (i.e., *0*) inducing the fault does not look different from other inputs under any structural criteria. In practice, this preceding situation may be a common situation for testers due to the wide use of black-box components. Therefore, appropriate test adequacy criteria specific to testing component-based software should be developed.

In this paper, we establish a suitable test adequacy criterion for testing software built on black-box components. Our basic idea is to define interface contracts for black-box components and use mutation operators defined on interface contracts to simulate faults possibly caused by the misunderstandings between the component provider and the system integrator. According to the notion of Design-by-

Contract [25], it is helpful to establish interface contracts between the provider and the user of a software component. When the source code of a component is not available, contracts defined on the interfaces of the reusable component can work as the "intermediary" between the component provider and the system integrator who reuse the component, because contracts can define the rights and the obligations of both providers and users to help distinguish their responsibility. Due to the nature of interface contracts, mismatches in component composition can be viewed as deviations from this intermediary by either side. Therefore, our approach uses mutants of interface contracts to simulate these deviations. Furthermore, we performed an experimental study on applying this criterion in two typical tasks in regression testing (i.e., test-suite reduction and test-case prioritization). The results confirm the effectiveness of our criterion.

The rest of this paper is organized as follows. Section 2 discusses research related to our approach. Section 3 presents the details of our approach. Section 4 describes an experimental study of the proposed criterion. Section 5 discusses the benefit and cost of using contracts. Section 6 presents conclusions.

2. Related Work

Some recent research [4,6,13,19,18] has focused on testing components. However, when testers need to test a software system built on black-box components, testing individual black-box components can ensure only the quality of these basic building blocks in the system. The quality of the entire system still cannot be ensured without adequate integration or system testing.

As a component-based system can be viewed as the integration of components, test adequacy criteria for integration testing are related to our research. Typically, these criteria can be classified as program-structure-based criteria, specification-based criteria, and design-information-based criteria.

Program-structure-based criteria focus on how the structure for integration has been covered during testing, including data-flow-based and control-flow-based criteria. Data flow analysis concerns with the definitions and uses of variables, and control flow analysis examines the execution of statements, conditions, and branches. Jin and Offutt [20, 27] developed a series of coupling-based criteria for integration testing. They defined a set of coupling-based test paths according to the definitions and uses of variables, and also proposed a set of testing criteria, including "call-coupling", "all-coupling-defs", "all-coupling-uses", and "all-coupling-paths", to help measure the effectiveness of a test suite. Harrold et al. [14] and Linnenkugel et al. [23] analyzed inter-procedural data- and control-flow information, and proposed integration testing criteria, such as "definition-use pairs" and "intraprocedural definition-pairs" [14], "all-paths", "all-edges", "all-nodes", and "all-p-uses" [23]. However, when source code is not available, the data-flow and control-flow information, which is essential to all the preceding structural criteria, cannot be easily acquired.

With the cost of constructing formal and complete specifications for the software under test, Ammann et al. [3] proposed a specification-based coverage criterion to evaluate test suites. Using the detailed specification defining the states and transitions for each class of the system under test, Gallagher et al. [12] acquired data dependences among classes and constructed data flow graphs for the system under test. Based on the coverage of data flow graphs, they proposed several criteria for integration testing. Recent research has shown that information derived from software design, such as collaboration diagrams and statecharts, can be used to define testing criteria that can help test data generation and selection [1,16]. However, detailed specifications or design documents of the applications (especially the reusable components) under test are often unavailable in testing of software built on black-box components.

Wu et al. [34] and Gao et al. [13] proposed a technique for testing component-based software based on the component interaction graph (CIG), which indicates the interactions and the dependence relationships among components. In particular, they proposed a family of test adequacy criteria, including all interfaces (AI), all events (AE), all context dependence relationships (ACD-1), and all content dependence relationships (ACD-2). However, when the source code of a component is not available, the content dependence relationship, which can be derived only from a class diagram, cannot always be acquired.

Delamaro et al. [8] developed the interface mutation (IM) approach that applies mutation testing in software integration testing. Their approach analyzes three types of faults that may occur during the interactions between procedures, and designs two groups of interface mutation operators to simulate these faults. The first group is applied inside an interface to mutate the source code of implementation of the interface, and the second group is designed to mutate the declaration and invocations of the interfaces. Therefore, the component source code needs to be available if we want to apply the first group of operators in testing software built on black-box components. Although also based on mutation testing, our approach aims at mutating interface contracts to support integration testing of component-based software built on black-box components.

From the preceding analysis, when facing software constructed with black-box components, the preceding criteria can hardly be fully applicable, as they more or less rely on the source code or detailed document of the components.

3. Interface-Contract Mutation (ICM) for Testing Component-Based Software

To establish a suitable criterion for testing software built on black-box components, we mutate the interface contracts of each black-box component. Our approach focus on the faults that could possibly happen when the component users integrated reusable components in their applications. Like other mutation testing approaches, such as traditional mutation [9,17,26], interface mutation [8,33], class or object mutation [2,21,24], specification mutation [5,11,31], and conception-model mutation in database applications [7], our approach uses mutation operators, which simulate various types of faults in component composition to generate mutants, and uses a mutation score, which defines the capability of killing mutants to measure the quality of test suites.

In Section 3.1, we analyze possible faults caused by the misunderstanding between the component providers and the system integrators. We design a set of mutation operators to simulate these faults in Section 3.2. Because our mutation is applied on the interface contracts, the mutation oracle used in traditional mutation testing is not applicable. As a result, we give an algorithm to detect interface-contract mutants in Section 3.3. The algorithm works as a mutation oracle. Based on the oracle, we define our contract mutation score in Section 3.4.

3.1. Fault Model

Because we are mainly concerned with faults in component composition, our fault model focuses on faults caused by the misunderstanding between system integrators and component providers. As interface contracts are the agreements that both providers and integrators should adhere to, the misunderstanding would result in deviations from the agreements by one or both sides. Therefore, the main objective of our fault model is to characterize how component providers and integrators can deviate from interface contracts. We next present four cases of deviations. The first two cases describe how integrators can deviate from the contracts, whereas the last two cases describe how component providers can deviate from the contracts.

(1) Precondition Weakening. A weaker precondition occurs when the constraints under which the component can perform correctly are wrongly broadened by the integrator. Thus, if the precondition of a component is weaker than it should be, the component may have to deal with more inputs than it actually can. This case simulates the situation that while reusing a component, the integrator passes to an interface some illegal inputs that cannot be correctly handled by the component providing the interface.

(2) Postcondition Strengthening. A stronger postcondition occurs when the integrator imposes stricter constraints on the outputs of the component. Thus, if the postcondition of a component is stronger than it should be, some correct outputs of the component may be viewed as incorrect by the integrator. This case simulates the situation that the integrator misunderstands the functionality of the component and treats some legal results as unacceptable when using some interfaces.

(3) Precondition Strengthening. A stronger precondition occurs when the provider imposes a stricter requirement on the component integrator who intends to reuse a component. Thus, if the precondition of a component is stronger than it should be, the component may not provide correct functionality when the inputs do not satisfy the strengthened precondition. This case simulates the situation that the component fails to handle some legal inputs as it is supposed to.

(4) Postcondition Weakening. A weaker postcondition occurs when the constraints that the outputs of the component should satisfy are wrongly broadened by the provider. Thus, if the postcondition of a component is weaker than it should be, the integrator has to face some illegal outputs from the component. This case simulates the situation that the component fails to always produce legal outputs as it is supposed to.

3.2. Interface-Contract Mutation Operators

We first translate interface contracts into conjunctive normal forms for the convenience of defining mutation operators (Section 3.2.1), and then describe the mutation operators, which are designed to simulate faults characterized by the preceding fault model (Section 3.2.2).

3.2.1. Normal Form for Contracts

In our new approach, we adopt the grammar of component-interface contracts defined in our previous work [19], and we do not repeat the defining grammar due to space limit. For the convenience of defining mutation operators for interface contracts, we transform preconditions and postconditions into the conjunctive normal form, whose definition is described below.

Definition 1. A contract is a pair $<C_{pre}, C_{post}>$ where C_{pre} is a precondition and C_{post} is a postcondition. A precondition or postcondition is in its conjunctive normal form if and only if it is in the form $C_i = \bigwedge_{0 \le j \le n} C_{ij}$, where C_{ij} is in disjunctive normal form $C_{ij} = \bigvee_{0 \le k \le s} C_{ijk}$. C_{ij} is called an *item* and C_{ijk} is called an *atomic expression*.

```
Algorithm  DetectMutants
Input    E: the execution trace of the test case t
         $M_{all}$: the set of interface-contract mutants of the components invoked in E
Output:  $M_{killed}$: the set of killed interface-contract mutants
begin
    for each invocation of an interface (denoted as $f_i$) of a component invoked in E do
        for each interface-contract mutant (denoted as $m_j$) in $M_{all}$ do
            if $m_j$ is a mutant produced by mutating the precondition of $f_i$
            then if the result of the mutated precondition differs from that of the original precondition
                then Remove $m_j$ from $M_{all}$; Add $m_j$ into $M_{killed}$
                end if
            else if $m_j$ is a mutant produced by mutating the postcondition of $f_i$
                then if the result of the mutated postcondition differs from that of the original postcondition
                    then Remove $m_j$ from $M_{all}$; Add $m_j$ into $M_{killed}$
                    end if
            end if
        end do
    end do
end
```

Figure 1. Algorithm *DetectMutants* for determining which mutants are killed by one test case

Because both the precondition and the postcondition of each contract are logical expressions, we can normalize them according to the rule of normalization of logical expressions. Consider the *TriTyp* [26] example, whose contract prescribes the calculation of the triangle type according to the three sides. The contract is as follows.

```
/*  @pre a>0;
    @pre b>0;
    @pre c>0;
    @post  Result ==3 implies (a==b && b==c);  */
    public int TriTyp ( int a, int b, int c)
```

The contract in the Javadoc form requires that the length of each side should be larger than *0*, and when the output is *3* (indicating an equilateral triangle), the inputs *a*, *b*, and *c* should be equal. Thus, we can transform the precondition as (1) and the postcondition as (2).

$$C_{pre} = a>0 \wedge b>0 \wedge c>0 \qquad (1)$$

$$C_{post} = (Result\ !=3 \vee a==b) \wedge (Result\ !=3 \vee b==c) \qquad (2)$$

3.2.2. Interface-contract Mutation Operators

Table 1 shows a set of interface-contract mutation operators (ICMOs) defined for the four cases in the fault model (Section 3.1). Each operator in the first column is designed to simulate the fault that is induced by the fault model showed in the second column. The third column describes the rules of implementing ICMOs. We can implement the rule of deleting a precondition/postcondition item by replacing the item with "*true*". Detailed rules of weakening and strengthening atomic expressions are listed in Table 2. Each ICMO should be used to mutate each contract of the interfaces of a black-box component.

The *Constant* in Table 2 is the minimum incremental value of Q's data type or is an incremental value that the tester has suggested. Because the *forall* and *exists* expressions can be translated to a group of expressions, these two expressions are not listed in the table. For example, "*forall x in {1, 3, 5} Result>0*" can be translated to "*!(x==1 || x==3 || x==5) || Result>0*".

Table 1. Interface-contract mutation operators

Name	Fault Model	Rules
PreWk	Precondition Weakening	Delete a precondition item
		Weaken an atomic expression
PreStr	Precondition Strengthening	Strengthen an atomic expression
PostWk	Postcondition Weakening	Delete a postcondition item
		Weaken an atomic expression
PostStr	Postcondition Strengthening	Strengthen an atomic expression

Table 2. Detailed mutation rules

Original Atomic Expression	Mutated Atomic Expression	
	Strengthening	Weakening
P==Q	------------------	P>=Q, P<=Q
P!=Q	P>Q, P<Q	--------------------
P>Q	P>Q +*Constant*	P>=Q, P!=Q
P<Q	P<Q - *Constant*	P<=Q, P!=Q
P>=Q	P>Q, P==Q	P>=Q - *Constant*
P<=Q	P<Q, P==Q	P<=Q + *Constant*

3.3. Interface-Contract Mutation Detection

In mutation testing, a mutation oracle, which is a person or a program to distinguish the original program from the mutant [22], is also a necessity besides the original program and its mutants. Figure 1 illustrates an algorithm (named *DetectMutants*) for determining which mutants are killed by a test case. The algorithm works as an oracle of our interface-contract mutation. In the algorithm, we record one execution trace for each test case, and for each execution trace we concentrate on each invocation of an interface of a component. Before the invocation, we check whether the result of any mutant of the precondition differs from that of the original precondition, which means the value of one of them is *true* and the other is *false*. For example, if we check a precondition $(a>0)\wedge(b<0)$ (denoted as P) and its mutant $(a>0)\wedge(b<=0)$ (denoted as M) using a test case ($a=1$, $b=0$) (denoted as t), the value of P is *false* and the value of M is *true*. Therefore, M is killed by t. Similar checking for mutants of postconditions is also performed after the invocation is returned. Based on *DetectMutants*, whether a mutant is killed by a test suite can be determined as follows. If a mutant is killed by at least one test case in a test suite, the mutant is said to be killed by the test suite.

Compared with traditional mutation testing, our interface-contract mutation using *DetectMutants* is a low-cost mutation testing approach for reducing the high cost of executing a large number of mutants. First, as our ICM mutates only the interface contracts, whose size is usually much smaller than that of source code, the number of mutants is significantly smaller than that of source-code mutation. Second, as mutating only interface contracts cannot affect the execution results of the original program, when detecting which mutants can be killed by a test case, we can instrument all mutated preconditions and postconditions in the program and evaluate all of them during one program execution with the test case.

3.4. Interface-Contract Mutation Score

The mutation score is the ultimate metric of test adequacy in mutation testing. In our approach, the definition of the mutation score is the same as traditional mutation testing. If we use N to denote the number of interface-contract mutants of a system S, N_E to denote the number of the equivalent interface-contract mutants that cannot be killed by any test case, and N_D to denote the number of the mutants killed by a test suite T, then the interface-contract mutation score MS is defined as:

$$MS(S, T) = N_D / (N - N_E) \qquad (3)$$

4. An Experimental Study

This section presents an experimental study that we conducted to evaluate the effectiveness of the proposed approach. We applied interface-contract mutation on three subjects (Section 4.1), and compared the experimental results of interface-contract mutation criterion with other criteria (Section 4.2). Finally we discuss some threats to the validity of our evaluation (Section 4.3).

4.1. Experimental Setup

We conducted our experimental study on three subject systems (Section 4.1.1), each of which has two groups of contracts and an initial test suite (Section 4.1.2). To evaluate the effectiveness of interface-contract mutation criterion, we compared it with IM and CIG-based criteria (Section 4.1.3) on test-suite reduction and test-case prioritization (Section 4.1.4).

Table 3. Summary information of subjects

Subjects	Triangle	ATM	Finance
Lines of Code	192	>4700	>5500
# Seeded Faults	5	12	9
# Atomic Expressions (Manually)	30	10	13
# Atomic Expressions (Daikon)	18	61	20

4.1.1. Subjects

In this experimental study, we use the *Triangle* system, the *ATM* system, and the *Finance* system as subjects. We suppose the source code of some parts of each system is not available to present various black-box components. The number of Lines of Code (LOC) and the number of faults seeded in the source code of each subject are summarized in Table 3.

The *Triangle* system can accept a group of integers as input, calculate the maximum, the minimum, and the middle value, and determine the triangle's type with the three values as sides. Some classical programs such as *Max*, *Min*, *Mid*, and *Trityp* [26] have been reused as reusable classes and they are the black-box components of *Triangle*.

The *ATM* system [1], which has been used in previous research of Yuan and Xie [35], simulates functions provided by an *ATM* machine and we treat *Balance Inquiry*, *Deposit*, *Withdrawal*, and *Transfer* as reusable classes without source code. Note that *ATM* used in this experimental study is not the one used in our previous work [19,18], which is made in-house and contains only about 300 lines of code.

The *Finance* system reuses interfaces provided by an open source Java library *MoneyJar.jar* [2], and its main program is based on *MoneyJar*'s example program *invoicer.java*. *Finance* can generate invoices including detailed charges and taxes for customers. APIs provided by *MoneyJar.jar*, which are used to compute calendars, charges, and taxes, are treated as black-box components.

4.1.2. Interface Contracts and Test Suites

Contracts are the agreements between the component provider and system integrator, and they can be defined manually or be inferred using a tool. *Daikon* [3] [10] is a dynamic invariant detector, which can report invariants at different program points in

[1] Source code and design documents can be downloaded from http://courses.knox.edu/cs292/ATMExample/Intro.html

[2] Source code and example programs can be downloaded from http://sourceforge.net/projects/moneyjar/

[3] The *Daikon* tool and user manual can be downloaded from http://pag.csail.mit.edu/daikon/

several formats. The DBC-format invariants (inferred by *Daikon*), which are generated at the entry and exit points of an interface, can be viewed as that interface's precondition and postcondition, respectively. Therefore, component providers can infer contracts for their components using a tool such as *Daikon*, and published these contracts together with their components. In our experimental study, each reused component interface has two contracts: one is specified manually and the other is inferred by *Daikon*. The test cases for *Daikon* are randomly generated according to the parameters' data type of the reused interfaces. For example, the *Triangle* system reuses two interfaces from the reusable classes, and one is the interface "*getTriTyp*", which can calculate the triangle type according to the input of three sides. The interface contract defined manually is shown in Figure 2 and the contract inferred by *Daikon* is listed in Figure 3.

For each subject, the numbers of atomic expressions (see Definition 1 in Section 3.2.1) included in these two kinds of contracts are listed in Table 3, respectively. Note that although *Daikon* often infers many invariants, it does not mean that the quality of the inferred contracts is always good. For example, for the *Deposit* interface of *ATM*, *Daikon* infers "Deposit_Amount != Account_Number". This invariant compares two integer parameters that are not comparable. Thus it is not helpful for testing.

To evaluate the preceding criteria, we manually generated an initial test suite for each subject based on the preconditions of their interface contracts, using equivalence-class partitioning and boundary-value analysis. The number of test cases and fault-exposure ratio (abbreviated as FER) for each test suite are listed in Table 4. The rows of Quantity, M-FER, C-FER and S-FER show the number of test cases, the fault-exposure ratios of the faults seeded in the main program, the components, and the entire system of each initial test suites, respectively. The fault-exposure ratio of *Finance* cannot achieve 100% because the test suite cannot reveal faults seeded in the code for the determination of "*TaxOnTax*". We checked the code of "*TaxOnTax*" manually and found that it is dead code, but we are not sure if there is other dead code in *Finance*.

4.1.3. Studied Criteria

We conducted our experimental study by comparing our approach with interface mutation (IM) [8,33] and techniques based on component interaction graphs (CIG), including All Interfaces (AI), All Events (AE), and All Context Dependences (ACD-1) [34, 13]. Because of the unavailability of source code of reused components, the first group of IM operators requiring source code was not applied, and the ACD-2 criterion using content dependence relationships derived from detailed design documents was not considered.

```
 /* @pre a>0 && b>0 && c>0;
    @post Result==1||Result==2||Result==3||Result==4;
    @post Result ==1 implies (a!=b && b!=c);
    @post Result ==2 implies (a==b || b==c || a==c);
    @post Result ==3 implies (a==b && b==c);
*/
    public int getTriTyp ( int a, int b, int c)
```

Figure 2. Interface contract defined manually

```
Tri.getTriTyp(int, int, int):::ENTER
a>= b
b>=c
a>=c
Tri.getTriTyp(int, int, int):::EXIT
return >= 1
return != orig(a)
return != orig(b)
return != orig(c)
```

Figure 3. Interface contract inferred by *Daikon*

Table 4. Initial test suites of subject systems

Subjects	Triangle	ATM	Finance
Quantity	183	79	120
M-FER① (%)	100.00	100.00	100.00
C-FER② (%)	100.00	100.00	75.00
S-FER③ (%)	100.00	100.00	77.78

①M-FER: the FER of the faults seeded in the main program
②C-FER: the FER of the faults seeded in components
③S-FER: the FER of the faults seeded in the entire system

4.1.4. Application Scenarios

Test-suite reduction and test-case prioritization are two typical scenarios of applying a test adequacy criterion in regression testing. We evaluated the effectiveness of applying the studied criteria in these two scenarios in our study. To be fair with all the five criteria, we used the algorithm proposed by Harrold et al. [15] in test-suite reduction, and used the strategy for achieving additional coverage in prioritization (Rothermel et al. [29] show different strategies in test-case prioritization). To evaluate the effectiveness of interface mutation (IM), Delamaro et al. [8] seeded faults in both components and their callers, and used fault-exposure ratios as the metric. Similar to IM, our ICM is also a mutation-based testing approach. Therefore, we seeded faults in the main program and source code of the reused components, and adopted fault-exposure ratios as the metric for test-suite reduction in our experimental study. We seeded faults in reused components because developers cannot ensure that there is no fault in published components, and some research has focused on testing black-box components [6,13,19,18]. The metric used in test-case prioritization is the average of the percentage of faults detected (APFD) metric proposed by Rothermel et al. [29]. We mutate some lines of source code using traditional mutation operators CRP, SVR, AOR, LCR, and ROR [26] to seed faults, including wrong constants and variables, faults in arithmetic, logical, and relational operators.

4.2. Result and Analysis

In this section, we report the experimental results of test-suite reduction (Section 4.2.1) and test-case prioritization (Section 4.2.2) using the interface-contract mutation (ICM), IM, AI, AE, and ACD-1 (denoted as ACD in this section). In order to distinguish the two groups of contracts inferred by *Daikon* and defined manually in our approach, we denote them as ICM_D and ICM, respectively.

4.2.1. Test-Suite Reduction

The major experimental results for test-suite reduction are listed in Table 5. The rows of Adequacy, Quantity, M-FER, C-FER, and S-FER show the scores of adequacy criteria, the number of test cases, the fault-exposure ratios of the faults seeded in the main program, the components, and the entire system using the reduced test suites, respectively. From the table, we can observe that all the studied criteria can help reduce the initial test suites significantly. Our ICM seems to be more effective in finding faults, as the exposure ratios of our ICM are always higher than those of the others for the three subjects. Actually, the test cases selected by our ICM can achieve most of the fault-exposure ratios of initial test suites, whereas the other criteria cannot. The research of Rothermel et al. [28] has shown that test-suite reduction may reduce the ability of exposing faults. In our study, the test cases selected by our ICM seem to be more likely to preserve the ability of exposing faults. For *Triangle* and *Finance*, the fault-exposure ratios of the test cases selected by our ICM are the same as those of the initial test suites. For *ATM*, although the fault-exposure ratio of the test cases selected by our ICM is not the same as the initial test suite, it is still higher than those of the test cases selected by other criteria.

Table 5 illustrates that our ICM_D is not as helpful as our ICM, because the fault-exposure ratio of our ICM_D is higher than that of IM, AI, AE, and ACD in the *Triangle* System but lower than AE and ACD in the *Finance* System. We analyzed the contracts inferred by *Daikon* and found that these contracts exclude some useful information that integrators may be quite interested in. These contracts are inferred from executions of each independent component or interface but not from executions of the integrated system. The relationships of the interface outputs and some variables in the main program are not considered. It indicates that contracts representing only component providers' requirements but not expressing integrators' requirements are not enough. In addition, the version of *Daikon* that we used can detect only invariants including three variables at most, so contracts that consider the relationships of more than three variables are omitted. At last, when an interface parameter is a class *X* whose private member *x* can be accessed only through a *getx()*method, or when a

Table 5. Comparison of selected test cases

Triangle System						
Criteria	IM	AI	AE	ACD	ICM_D	ICM
Adequacy①(%)	100.00	100.00	85.77	75.00	100.00	98.53
Quantity	6	1	2	2	4	8
M-FER (%)	0.00	100.00	100.00	100.00	100.00	100.00
C-FER (%)	50.00	50.00	50.00	50.00	75.00	100.00
S-FER (%)	40.00	60.00	60.00	60.00	80.00	100.00
ATM System						
Criteria	IM	AI	AE	ACD	ICM_D	ICM
Adequacy (%)	91.45	100.00	100.00	85.00	71.11	85.56
Quantity	4	2	3	8	5	6
M-FER (%)	66.67	66.67	66.67	66.67	50.00	66.67
C-FER (%)	33.33	33.33	50.00	50.00	66.67	66.67
S-FER (%)	50.00	50.00	58.33	58.33	58.33	66.67
Finance System						
Criteria	IM	AI	AE	ACD	ICM_D	ICM
Adequacy (%)	83.87	100.00	100.00	100.00	60.00	71.43
Quantity	1	1	2	4	3	4
M-FER (%)	100.00	100.00	100.00	100.00	100.00	100.00
C-FER (%)	12.50	37.50	50.00	62.50	37.50	75.00
S-FER (%)	22.22	44.44	55.56	66.67	44.44	77.78

①Adequacy: Adequacy stands for mutation scores (MS) in IM and ICM; Adequacy stands for coverage in AI, AE, and ACD (see CIG approaches in Section 2.2 and Section 4.1.3.

Table 6. Mutants quantity comparison

Approach		Mutants	Killed Mutants	Equivalent Mutants
Triangle	IM	95	84	8
	ICM_D	48	24	24
	ICM	101	67	33
ATM	IM	120	107	3
	ICM_D	73	32	28
	ICM	27	18	6
Finance	IM	31	26	0
	ICM_D	27	12	7
	ICM	17	10	3

component provides some query interfaces, such as a *top()* method of a *Stack* component, testers can heuristically write X.getx() or top() in contracts, but Daikon cannot. All these limitations may cause ICM_D not to be as effective as ICM.

Note that actual capabilities for exposing faults of our ICM criteria are considerable when their reduced test suites cannot achieve very high contract mutation scores. This observation further confirms that our ICM seems to be more practical in exposing faults than IM and CIG-based criteria as an adequacy criterion for testing systems built on black-box components.

Because ICM, ICM_D, and IM are mutation-based criteria, we performed a further comparison of them in Table 6. The research results of Vincenzi et al. [33] show that the computational cost of IM is not greatly reduced compared to traditional mutation testing. The reason is that the first group of mutation operators, which are applied to mutate the implementation of interfaces, can generate thousands of mutants. As our experiments have not applied these IM operators, the number of IM mutants is small. Table 6 shows that the number of ICM mutants is almost as small as or much smaller than that of IM mutants in all systems. This observation indicates that all the mutation techniques are efficient in this context. Furthermore, all the mutation techniques produced a group of equivalent mutants in our experiment. This result indicates that more efforts for identifying equivalent mutants would be needed in both IM and our ICM (as well as ICM_D).

4.2.2. Test-Case Prioritization

The results for test-case prioritization are shown in Figures 4-6. These figures show changes of the percentage of faults detected (the vertical axis) corresponding to the increase of the number of test cases (the horizontal axis). Due to space limit, the figures do not show the curves after achieving highest fault-exposure ratios. Figures 4-6 show the results of exposing faults in the entire subject systems. The results in all these figures confirm that our ICM and ICM_D can usually achieve better fault-exposure ratios than other criteria with the same number of test cases. We can also observe that our ICM or ICM_D always first achieves the highest fault-exposure ratios in the three subjects.

In Figures 4-5, the number of test cases for achieving 100% FER using ICM in *Triangle* and *ATM* is much larger than that in test-suite reduction, and we suspect that it is probably because our interface-contract mutation operators produced easily-killed mutants. When prioritizing test cases according to the mutation score (MS), once MS achieves its highest value early, the remaining test cases would not be prioritized.

4.3. Threat to Validity

Threats to internal validity mainly include factors that may also be responsible for experimental results except for the factors studied in the experiments. In our experiments, the algorithm of test-suite reduction, the strategy and the metric of test-case prioritization, and the metric for evaluating the quality of a test suite may affect our experimental results. In order to reduce this threat, we adopted algorithms, strategies, and metrics from previous work. We used the test-suite reduction algorithm proposed by Harrold et al. [15], the additional coverage strategy and the APFD metric for test-case prioritization proposed by Rothermel et al. [29], and the fault-seeding approach and the fault-exposure-ratio metric adopted by Delamaro et al. [8].

One issue raised in the experiments is that there is no powerful evidence for the reliability of using fault-exposure ratios as metrics. Thus we are not very sure if faults seeded in the main program and components' source code could precisely simulate faults appearing in practice. However, we have not found more realistic techniques for validating the effectiveness of test suites in testing component-based software. So we

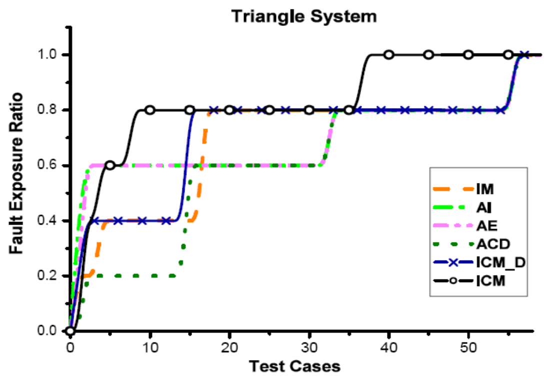

Figure 4. System-fault exposure on Triangle System

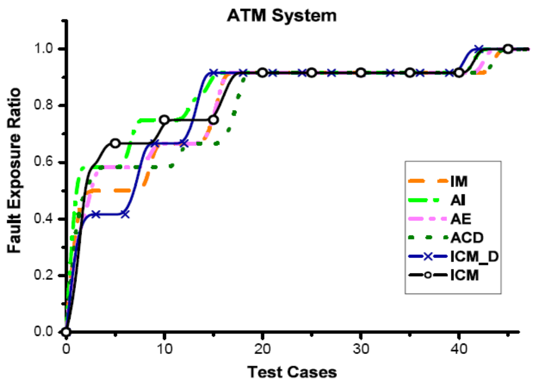

Figure 5. System-fault exposure on ATM System

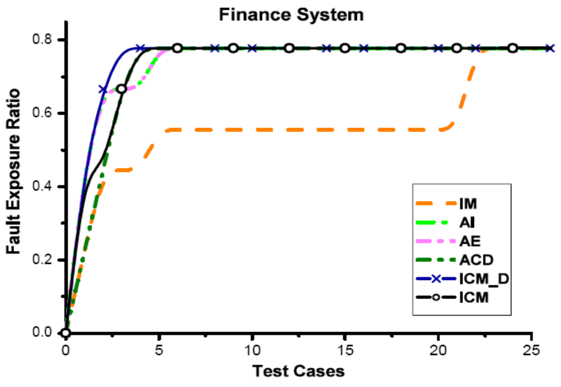

Figure 6. System-fault exposure on Finance System

used fault-exposure ratios as metrics, which have been widely used in testing and are also adopted in interface mutation [8]. Because our fault model is based on the deviation of contracts, we expect that our approach could perform better than what the experimental results have shown if the seeded faults can precisely simulate faults appearing in testing software built on black-box components in practice.

Factors that may affect the generalization of experimental results are threats to external validity. In our study, we used only one small subject and two medium-sized subjects built on Java reusable class libraries. Although these subjects are programs from public software libraries or used in previous work, our experimental results may not be generalized to other different programs. To reduce this threat, we plan to do more experiments with large-sized systems that reuse more kinds of black-box components in our future work, such as large distributed systems built on EJBs and Web Services.

5. Benefits and Costs of Using Interface Contracts

The main reason for the effectiveness of interface-contract mutation lies in the use of interface contracts. Interfaces of black-box components alone contain little information of the components, whereas interface contracts can provide the tester with more valuable information. Another benefit of interface contracts is that they themselves can help reveal faults, because interface contracts can capture invalid inputs and outputs between the components and the main program.

The preceding benefits are achieved with costs of specifying interface contracts, which are essential for the use of interface-contract mutation. However, contracts are not defined only for the purpose of providing information for testing. Meyer [25] pointed out that contracts can help programmers write high-quality programs, and Szyperski [32] also pointed out that contracts can make reusable building blocks easier to implement and compose. Therefore, the effort for defining contracts is not just the cost of providing information for testing, but actually the expense paid to achieve all the preceding advantages. Consequently, interface-contract mutation can be considered as a low-cost approach in this context.

6. Conclusions

Because of unavailability of source code, testing software built on black-box components becomes a challenging issue. In our research, we proposed a new adequacy criterion for testing this type of software. We performed an experimental study on our criterion together with four other previously proposed criteria (IM was adapted in order to test this type of software in our study) for two typical tasks in regression testing. The experimental results show that our criterion is more effective for both test-suite reduction and test-case prioritization than previous criteria.

We have applied contract mutation in testing black-box components [19,18] ([18] is the journal version of [19] with two more subject components in its experimental study). Compared with previous work, the main contributions of our work are as follows:

- This paper aims to address a new problem. In our previous work, we applied contract mutation to test black-box components. However, when testers face a software system built on black-box components, testing individual components can ensure only the quality of these building blocks in the system. Testers still need to ensure the quality of the entire system. In this paper, we aim at testing software built on black-box components. Specifically, the primary concern in this paper is the misunderstanding between the component provider and the system integrator.
- This paper proposes a fault model and mutation operators that are specific for testing component-based software. The mutation operators in our previous work simulated only the deviation from the interface contracts by the component provider. When composing a system using components, both the component provider and the system integrator may deviate from the interface contracts. Therefore, in this paper, we analyzed the fault model for deviations by both of them, and proposed four mutation operators to simulate them. Among the operators, *PreStr* and *PostWk* are totally new. Furthermore, the mutation operators in this paper are based on the normal form of contracts that are easier to implement.
- The evaluation in this paper uses medium-sized systems. We have used an in-house version of *ATM* and *Tritype* in the evaluation of our initial research on testing black-box components [19], and used two more programs (i.e., *Middle* and a Siemens program named *Tcas*) in its extension [18]. All these programs are small-sized subjects whose LOCs are from 24 to 300. In this paper, we evaluated our approach by conducting an experimental study on three subjects, two of which are medium-sized programs with more than 4700 and 5400 lines of code, respectively.
- This paper also evaluates the effectiveness of our ICM approach on automatically inferred contracts. In the experimental study of our previous work, we also considered manually defined contracts for components interfaces. In this paper, we not only manually specified contracts for the subjects, but also used a dynamic invariant detector (i.e., *Daikon*) to infer contracts for reused components. Note that mutating automatically inferred contracts provides a potential solution to further automate our technique and reducing the cost of defining contracts.
- The evaluation in this paper is based on comparison with existing criteria for typical tasks in regression testing. In the experimental study of our previous work, we compared the contract-based mutation operators with the five key traditional mutation operators. In this paper, we compared our ICM with other test adequacy criteria, including interface mutation and CIG-based criteria. Furthermore, this paper evaluates our approach in two important and practical application scenarios of test adequacy criteria in regression testing: test-suite reduction and test-case prioritization.

In practice, many components are stateful and legal operation sequences of components can help prevent these components from reaching error states. Currently, our approach does not support specifying legal operation sequences in contracts. We plan to use “trace” [25], which aims to define operation sequences of classes in our interface contract. We also plan to develop mutation operators for “trace” in our future work.

7. Acknowledgements

This research is sponsored by the National 973 Key Basic Research and Development Program No. 2002CB312003 and the National Science Foundation of China No. 90412011and No.60403015.

8. References

[1] A. Abdurazik, J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", *Proc. UML*, 2000, pp. 383-395.

[2] R.T. Alexander, J.M. Bieman, S. Ghosh, J. Bixia, "Mutation of Java Objects", *Proc. ISSRE*, 2002, pp.341-351.

[3] P. Ammann, P. Black, "A Specification-Based Coverage Metric to Evaluate Test Sets", *Proc. HASE*, 1999, pp. 239-248.

[4] S. Beydeda, V. Gruhn, "State of The Art in Testing components", *Proc. QSIC, 2003*, pp. 146-153.

[5] P. Black, V. Okun, Y. Yesha, "Mutation Operators for Specifications", *Proc. ASE*, 2000, pp. 81–89.

[6] L.C. Briand, Y. Labiche, M. Sówka, "Automated, Contract-based User Testing of Commercial-Off-The-Shelf Components", *Proc. ICSE*, 2006, pp.92-101.

[7] W.K. Chan, S.C. Cheung, T.H. Tse, "Fault-Based Testing of Database Application Programs with Conceptual Data Model", *Proc. QSIC*, 2005, pp.187-196.

[8] M.E. Delamaro, J.C. Maldonad, A.P. Mathur, "Interface Mutation: An Approach for Integration Testing", *IEEE Transactions on Software Engineering*, 27(3), 2001, pp.228-247.

[9] R.A. DeMillo, R.J. Lipton, F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", *IEEE Computer*, 11(4), 1978, pp. 34-41.

[10] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution", *IEEE Transactions on Software Engineering*, 27(2), 2001, pp.99-123.

[11] S.C.P.F. Fabbri, J.C. Maldonado, P.C. Masiero, M.E. Delamaro, "Mutation Analysis Testing for Finite State Machines", *Proc. ISSRE*, 1994, pp 220-229.

[12] L. Gallagher, A.J. Offutt, A. Cincotta, "Integration Testing of Object-Oriented Components Using Finite State Machines", *Software Testing, Verification, and Reliability*, 16(4), 2006, pp.215-266.

[13] J. Gao, H.J. Tsao, Y. Wu, *Testing and Quality Assurance for Component-Based Software*, Artech House, Boston, London, 2003, ISBN 1580534805.

[14] M.J. Harrold, M.L. Soffa, "Selecting and Using Data for Integration Testing," *IEEE Software*, 8(2), 1991, pp.58-65.

[15] M.J. Harrold, R. Gupta, M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite", *ACM Transactions on Software Engineering and Methodology*, 2(3), 1993, pp.270-285.

[16] J. Hartmann, C. Imoberdorf, M. Meisinger, "UML-Based Integration Testing", *Proc. ISSTA*, 2000, pp. 60-70.

[17] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets", *IEEE Transactions on Software Engineering*, 8(4), 1982, pp. 371-379.

[18] Y. Jiang, S.S. Hou, J.H Shan, L. Zhang, B. Xie, "An Approach to Testing Black-Box Components Using Contract Mutation", accepted by *International Journal of Software Engineering and Knowledge Engineering*, 2006

[19] Y. Jiang, S.S. Hou, J.H Shan, L. Zhang, B. Xie, "Contract-Based Mutation for Testing Components", *Proc. ICSM*, 2005, pp.483-492.

[20] Z. Jin, A.J. Offutt, "Coupling-based Criteria for Integration Testing", *Software Testing, Verification, and Reliability*, 8(3), 1998, pp.133-154.

[21] S. Kim, J. Clark, J. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs", *Proc. FMES*, 2000.

[22] G. Kovacs, Z. Pap, D.L. Viet, H.C. Wu, G. Csopaki, "Applying Mutation Analysis to SDL Specifications", *Proc. SDL*, 2003, pp.269-284.

[23] U. Linnenkugel, M. Mullerburg, "Test Data Selection Criteria for (Software) Integration Testing", *Proc. International Conference on Systems Integration*, 1990, pp. 709-717.

[24] Y.S. Ma, Y.R. Kwon, A.J. Offutt, "Inter-Class Mutation Operators for Java", *Proc. ISSRE*, 2002, pp.352-363.

[25] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, New York, second edition, 1997.

[26] A.J. Offutt, G. Rothermel, C. Zpf, "An Experimental Evaluation of Selective Mutation", *Proc. ICSE*, 1993, pp.100-107.

[27] A.J. Offutt, A. Abdurazik, R.T. Alexander, "An Analysis Tool for Coupling-based Integration Testing", *Proc. ICECCS*, 2000, pp.172 -178.

[28] G. Rothermel, M.J. Harrold, J. V. Ronne, C. Hong, "Empirical Studies of Test-Suite Reduction", *Software Testing, Verification and Reliability*, 12(4), 2002, pp. 219-249.

[29] G. Rothermel, R.J. Untch, C. Chu, "Prioritizing Test Cases For Regression Testing", *IEEE Transactions on Software Engineering*, 27(10), 2001, pp. 929-948.

[30] G. Rothermel, R.H. Untch, C.Chu, M.J. Harrold, "Test case prioritization: an empirical study", *Proc. ICSM*, 1999, pp.179-188.

[31] T. Sugeta, J.C. Maldonado, W.E. Wong, "Mutation Testing Applied to Validate SDL Specifications ", *Proc. IFIP*, 2004. pp. 193-208.

[32] C. Szyperski, "Components and Architecture", *Software Development*, 8(10), 2001 .

[33] A.M.R. Vincenzi, J.C. Maldonado, E.F. Barbosa, M.E. Delamaro, "Unit and integration testing strategies for C programs using mutation", *Software Testing, Verification, and Reliability*, 11(3), 2001, pp.249- 268.

[34] Y. Wu, D. Pan, M. Chen, "Techniques for Testing Component-Based Software", *Proc. ICECCS*, 2001, pp.222-232.

[35] H. Yuan, T. Xie, "Substra: A Framework for Automatic Generation of Integration Tests", *Proc. AST*, 2006, pp. 64-70.