Volume 81, issue 2, February 2008
ISSN 0164-1212
ELSEVIER
The Journal of Systems and Software
Special Issue: Model-Based Software Testing

Available online at www.sciencedirect.com

ScienceDirect

The Journal of Systems and Software 81 (2008) 234–248

The Journal of Systems and Software

www.elsevier.com/locate/jss

A relation-based method combining functional and structural testing for test case generation ☆

Shaoying Liu *, Yuting Chen

Hosei University, Tokyo, Japan

Available online 7 June 2007

Abstract

Specification-based (or functional) testing enables us to detect errors in the implementation of functions defined in specifications, but since specifications are often incomplete in practice for some reasons (e.g., lack of ideas, no time to write), it is unlikely to be sufficient for testing all parts of corresponding programs. On the other hand, implementation-based (or structural) testing focuses on the examination of program structures, which allows us to test all parts of the programs, but may not be effective to show whether the programs properly implement the corresponding specifications. To perform a comprehensive testing of a program in practice, it is important to adopt both specification-based and implementation-based testing. In this paper we describe a *relation-based test method* that combines the specification-based and the implementation-based testing approaches. We establish a set of relations for test case generation, illustrate how the method is used with an example, and investigate the effectiveness and weakness of the method through an experiment on testing a software tool system.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Software testing; Specification-based testing; Relationship-based testing

1. Introduction

Specification-based testing (SBT) of programs advocates that test cases are generated from specifications, because specifications are supposed to define the behaviors of corresponding programs. SBT has been extensively researched under different names, such as specification-based testing (Bernot et al., 1991), model-based testing (El-Far and Whittaker, 2001), and functional testing or black-box testing (Beizer, 1995), and its scientific nature and tool supportability have been significantly improved when formal specifications are employed. An obvious advantage of SBT over implementation-based (or structural) testing is that tests can be performed without the need to analyze program structures. This advantage becomes especially useful when testing component-based programs where the source code of the components may not be available.

There exist however many challenges in the application of SBT, some of which are caused by practical constraints, while others are due to theoretical immaturity. Firstly, it is often the case in practice that the specification of a program system is incomplete in defining the behaviors of the system. This problem may arise because of the lack of sufficient time, budget, or expertise for completing the specification before implementation; it may also be resulted from the situation where both the client and the analyst (who is responsible for writing the specification) have little knowledge in the phase of requirements analysis about what should exactly be done by the program. As a result, many new functions may be added and the original functions defined in the specification may be changed during the coding phase without updating the specification. All of these situations are highly likely to occur in industrial

☆ This work is supported in part by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research (No. 18500027) and by Hosei University under a special research grant for the year 2006.
* Corresponding author.
E-mail addresses: sliu@hosei.ac.jp (S. Liu), chenyt@st2.hosei.ac.jp (Y. Chen).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.05.036

projects since most of software projects in practice are carried out evolutionarily and incrementally according to a survey reported by Larman and Basili (2003). Consequently, the specification may become unsuitable for generating test cases and expected test results to test the program properly. Secondly, the interface of an operation defined in the specification may be altered in the program because the function of the operation may have to be implemented in a specific program structure for efficiency or maintainability, or because its function has to be modified to suit the implementation of additionally required functions. For these reasons, the test cases and expected results generated based on the interface of the operation in the specification may not be easily used to execute the program and analyze the actual test results. Thirdly, the attributes of variables in the specification, such as variable names, types, and syntactic representations, may not be preserved in the program. For example, the specification may be written in the *Z* notation (Woodcock and Davies, 1996), while the program is implemented in Java. Thus, test cases generated from the specification will have to be properly mapped to an appropriate syntactic representations of relevant variables in order that they can be used to execute the program. Since a correct mapping of this kind requires a thorough understanding of the corresponding parts in both the specification and the program as well as time and labor cost, performing the mapping will likely set an additional barrier for the application of SBT in practice.

To address the challenges mentioned above, in this paper we describe a novel and practical method for program testing that combines specification-based and implementation-based testing. In our method, formal specification is used as a technique to model and define the program under development, and as a reference document to help the tester (who is responsible for testing) determine whether the test cases generated cover all functional scenarios defined in the specification. By covering a functional scenario with a test case we mean that the test case can select the scenario for testing its implementation in the program. The actual generation of test cases for a test are carried out based on the interfaces of operations in the program, which is independent of both the specification structure and the program structure, but the knowledge of both specification and program structures may be utilized to select test cases and the desired structure coverages may be used as criteria to determine when the test can terminate. The principle underlying the test method is similar to that of the *grey box testing* that emphasizes the blending of functional and structural testing (Kaner, 2000). The question is how to generate test cases from an operation interface.

We describe a relation-based method in Section 3 by which test cases are generated based on various relations among all the input variables (including parameters and state variables). The relations are built in advance among data types of the programming language in which the program under testing is written, and they will be selectively applied to generate test cases. Selecting relations must guarantee that the relations are applicable to generating usable test cases (e.g., choosing the "great" than relation $>$ rather than the membership relation $\in$ between the two input integers x and y of an operation). Apart from this criterion, the tester can select relations at random, or in any way of his or her preference (e.g., choose relations between any two variables, and then extend to more variables if the current test cases generated are not sufficient).

When the "greater than" relation $>$, for example, is selected to apply to two input integer variables x and y of an operation to generate test cases, choosing test values to x and y from their types, respectively, may be based on different considerations. One is to consider the program structure (e.g., conditions $x = y + 5$, $x = 20 \wedge y = 18$); that is, values for x and y to meet the relation $x > y$ can be generated based on the analysis of the conditions in the program. Another consideration is the operation specification (e.g., pre- and postconditions, as detailed in Section 2). If neither the program nor the specification is considered, the types of x and y can be used as the only reference to choose their values. In general, taking program and specification structures into account decreases the capability of automating test case generation, but may enhance the effectiveness of selecting test cases. A test process can terminate if the following two conditions are both satisfied: (1) the test cases generated cover all the functional scenarios in the specification and (2) the test cases meet a specified coverage criterion in the program. We require that the test cases cover all functional scenarios in the specification because we expect to test whether all the functional scenarios are correctly implemented in the program. We can in principle choose any coverage criterion in the program (e.g., statement-coverage, branch-coverage, condition-coverage, path-coverage, or their combinations) (Beizer, 1990; Shannon et al., 2004), depending on the cost-effectiveness requirement or the criticality of the program.

Since there may be a gap between the program and its specification, a test meeting any coverage criterion in the program may not ensure to cover all the scenarios defined in the specification (e.g., some functions defined in the specification are not implemented in the program for some reason). It is also possible that the program implements more functions than those defined in the specification. If the test does expose such a gap, a justification must be made by all the parties concerned (e.g., the analyst, the programmer, and the tester) to explain why the gap exists.

The rest of the paper is organized as follows. Section 2 introduces basic concepts we use to prepare for the subsequent discussions in the paper. Section 3 presents a relation-based method for test case generation. Section 4 discusses the process for testing using the method. In Section 5 we illustrate how the method and the process are actually used with an example, and in Section 6 we describe an experiment on testing a software support tool system for the assessment of the test method. In Section 7 we discuss the related work, and finally, we conclude the paper and point out future research directions in Section 8.

2. Basic concepts

The discussions throughout the paper are based on the following assumptions: an operation is specified with pre- and post-conditions and the implementation of the operation preserves the variables used in the specification. Considering our expertise and the comprehensibility of specifications, we adopt the syntax in SOFL (Structured Object-Oriented Formal Language) (Liu, 2004) for operation specifications. The format of an operation specification is as follows:

process $A(input)$ *output*
ext *external variables*
pre A_{pre}
post A_{post}
end_process

where all the words in bold font are reserved words with intuitive meaning. For example, **process** indicates the start of an operation specification. Operation A takes the *input* together with some of the *external variables* that satisfy the precondition A_{pre}, and generates the *output* and updates some of the *external variables* that satisfy the postcondition A. The A_{post} defines a relation between the initial state and the final state before and after the operation, therefore it may involve both the initial external variables (e.g., $\tilde{x}$) and final external variables (e.g., x).

For the simplicity of presentation, we sometimes use the concise representation of an operation specification: A: $[A_{iv}; A_{ov}; A_{ev}; A_{pre}; A_{post}]$, where A_{iv} denotes the set of all the input variables, A_{ov} the set of all the output variables, and A_{ev} the set of all the related external variables, A_{pre} and A_{post} represent the pre- and postconditions of A, respectively.

The essential notions we need for discussions in the paper include *test case*, *test set*, *test*, and *functional scenarios* in the specification.

Definition 1. Let A:$[A_{iv}; A_{ov}; A_{ev}; A_{pre}; A_{post}]$ be an operation; $A_{iv} = \{x_1, x_2, \ldots, x_n\}$; and $A_{ev} = \{y_1, y_2, \ldots, y_m\}$. A test case for A is a group of values $v_1, v_2, \ldots, v_{n+m}$ bound to $x_1, x_2, \ldots, x_n, \tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_m$, respectively.

A test case for operation A is a group of values for its input variables and initial external variables. For example, suppose the operation *Add_Div* is defined as follows:

$$\begin{aligned} &Add\_Div : [\{x\}; \{\,\}; \{y\};\\ &\qquad x \neq 0;\\ &x > \tilde{y} \wedge (y = \tilde{y} + x \vee \tilde{y} > y) \vee x \leqslant \tilde{y} \wedge y = \tilde{y}/x], \end{aligned}$$

where all the variables are of the integer type.

Definition 2. A test set for operation A is a collection of test cases.

Table 1 shows a test set including three test cases.

Table 1
Examples of test cases

x	$\tilde{y}$
5	0
0	5
−100	−200

Definition 3. A test for operation A consists of a test set, a set of expected test results, and a set of corresponding actual test results.

For example, Table 2 extends Table 1 into a test, where the actual results are assumed to be produced by executing the corresponding program.

Definition 4. Let A be an operation and $A_{pre} \wedge A_{post} \equiv (A_{pre} \wedge C_1 \wedge D_1) \vee (A_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (A_{pre} \wedge C_n \wedge D_n)$, where C_i $(i \in \{1, \ldots, n\})$ is a guard condition and D_i a defining condition. Then, a functional scenario f_s of A is a conjunction $A_{pre} \wedge C_i \wedge D_i$, and such a form of operation A is called a functional scenario form or FSF for short.

A guard condition of operation A is a predicate in its postcondition that does not contain any output or final external variables of A, while a defining condition of A definitely contains some output or final external variables. A functional scenario describes an independent functional behavior under a certain circumstance, since the truth of each functional scenario leads to the truth of the whole FSF. Consider the operation *Add_Div* as an example. Its FSF consists of the two functional scenarios:

(1) $x \neq 0 \wedge x > \tilde{y} \wedge (y = \tilde{y} + x \vee y > \tilde{y})$.
(2) $x \neq 0 \wedge x \leqslant \tilde{y} \wedge y = \tilde{y}/x$.

A FSF is similar to a disjunctive normal form in the first order logic if the operation specification is deterministic; otherwise, it may be different. For example, the above operation *Add_Div* is a nondeterministic specification because when $x > \tilde{y}, y$ can be defined either by the expression $y = \tilde{y} + x$ or $y > \tilde{y}$. Apparently, the FSF is not a disjunctive normal form because functional scenario (1) contains a disjunction.

We have worked out an algorithm in our previous work (Liu et al., 2005) to convert any operation specification into an equivalent FSF. The essential principle of the algorithm is first to transform the conjunction of the pre- and

Table 2
An example of test

x	$\tilde{y}$	y (expected results)	y (actual results)
5	0	5	5
0	5	Any result	5
−100	−200	2	2

postconditions of the operation into an equivalent disjunctive normal form, and then transform the disjunctive normal form into an equivalent FSF by forming guard conditions and defining conditions. A guard condition is formed by grouping all the conditions in a disjunctive clause, which contains no output or final external variables of the operation, into a conjunction. A defining condition is formed by grouping conditions containing output or final external variables into a disjunction if they are in conjunction with the same guard condition. Since this algorithm is not our contribution in this paper and also beyond the scope of this paper, we omit its detailed introduction here.

With the concepts defined above, we define a criteria for SBT.

Criterion 2.1(*functional scenario coverage*) Let A be an operation and its FSF be $(A_{pre} \wedge C_1 \wedge D_1) \vee (A_{pre} \wedge C_2 \wedge D_2) \vee \cdots \vee (A_{pre} \wedge C_n \wedge D_n)$. Let T_A be a test set for A. Then, T_A must cover every functional scenario $A_{pre} \wedge C_i \wedge D_i$ $(i \in \{1, \ldots, n\})$.

The functional scenario coverage criterion emphasizes that every functional scenario defined in the specification needs to be covered by a test set, because such a test set will help to explore whether all the defined functional scenarios are implemented properly in the program. If the test set T_A makes $A_{pre} \wedge C_i$ $(i = 1, 2, \ldots, n)$ true (at least once), we say that T_A covers the scenario $A_{pre} \wedge C_i \wedge D_i$. Note that the scenario coverage criterion does not require the test set to cover all the desired functional scenarios (some of which may not defined yet in the specification for some practical reasons), but tries to ensure that the test set covers every defined functional scenario.

As described in Section 1, generating test cases directly from specifications faces many challenges in practice. In our testing approach, we mainly treat the specification as a reference for judging whether the functional scenario coverage is satisfied or not, but test cases are actually generated based on the interface of the operation using a particular technique called *relation-based method*, as discussed in detail in Section 3.

3. Relation-based method

The essential idea of the relation-based method is to generate test cases based on certain pre-established relations among input variables or variables and constants of an operation. The relations are established in advance based on certain universal principles, as discussed in detail from Section 3.1 to Section 3.5. When a specific operation is tested, the pre-established relations are applied selectively to generate test cases. Since the relations are all sufficiently simple, their application for test case generation can be supported automatically. Another strength of the method is that it allows relatively effective test case generation based only on the interface of the operation under test, without the need for the tester to analyze thoroughly either the corresponding specification or program.

3.1. Relations for test case generation

The strategy for test case generation based on relations results from the consideration that a program usually uses its input variables for conditions and/or decisions in its algorithm, and changes of the relations of their corresponding values are likely to cause changes of the evaluation results of the conditions and/or decisions. The input variables may be used directly (i.e., the variables appear in conditions or decisions as they are) or indirectly through propagated variables (e.g., the local variable y is initialized using the input variable x and y is used in some conditions or decisions in the program). Consider the operation *Add_Div* defined before as an example. Suppose it is implemented as a method in Java:

```
public double Add_Div (int x) {
if (x! = 0)
   if (x > y)
      y = y + x;
   else y = y/x;
else System.out.println("The precondition is not satisfied");
return y;
}
```

where variable y is an instance variable (or state variable). Obviously, the input variables x and y are used directly for decisions (e.g., $x! = 0, x > y$). If we select values for x and y so that they first satisfy the relation $x > y$ and then $x < y$ when x is not zero, it will make the decision $x > y$ in the inner *if–else* statement evaluate to true and false, respectively. When x is zero, that is, x satisfies the relation $x = 0$, it will cause the decision $x! = 0$ in the outer *if–else* statement false, which is different from the evaluation result of the first case.

This example shows that by selectively applying different relations among input variables or variables and constants we are likely to produce test cases to traverse certain parts of the program. The essential issue here is what relations we should build in order to produce effective test cases and meanwhile support the automation of test case generation.

We define relations over basic data types (e.g., integer, real) and data structures (e.g., array, object), and relations among different types or structures. In principle, an infinite number of relations can be built between some data types. For example, $x = y + \sigma$ represents an infinite number of relations between two integers x and y because the increment σ changes in the integer type; whenever σ changes, $x = y + \sigma$ represents a new relation between x and y. To convey the major idea of the relation-based method within the limited space in the paper, we only discuss some commonly used relations as examples. Readers who are interested in applying the method can extend or modify the relations introduced in this section to form their own relations for testing their programs.

Table 3
Relations between variables of the same or compatible types

Basic relations	Complex relations	Boundary relations
$x = y + \sigma$ where	x (or y) $> 0 \wedge x = y + \sigma$	x (or y) $= \mathit{Maximum}$
$\sigma\{=,>,<\}0$	x (and y) $> 0 \wedge x = y + \sigma$	x (or y) $= \mathit{Minimum}$
	x (or y) $= 0 \wedge x = y + \sigma$	x and $y = \mathit{Maximum}$
	x (or y) $< 0 \wedge x = y + \sigma$	x and $y = \mathit{Minimum}$
	x (and y) $< 0 \wedge x = y + \sigma$	$x = \mathit{Maximum} \wedge y = \mathit{Minimum}$
	$x = 0 \wedge y = 0$	$x = \mathit{Maximum} \wedge y = \mathit{Maximum}$

3.2. Relations over numeric types

The numeric types we use for implementation in this paper include *int* and *double*, and they are assumed to implement, respectively, the corresponding integer and real number types *int* and *real* in the specification. The discussions on these two types can be easily extended to other numeric types. Let x and y be two variables of numeric types. Consider the necessity of testing normal conditions, complex conditions, and boundary conditions, we build three groups of relations between x and y in Table 3, each dealing with a specific situation.

Since σ can be zero, positive, and negative, the basic relations for all kinds of comparisons between x and y actually contain rich relations, including $x > y$, $x < y$, and $x = y$. By changing the value of σ, we can build more relations between x and y. The complex relations show the basic relations between x and y under certain constraints on one of x and y or on both of them. The boundary relations allow to generate test cases that are not usually used in the execution of the operation. The Maximum and Minimum mean the maximum and minimum numbers of the machine on which the operation is executed, which may vary from machine to machine.

3.3. Relations on compound types

Compound types include array and linked list in most programming languages, and vector, set, sequence, and map in some recently developed languages (e.g., Java, C#). For the sake of space, we focus our discussion on the numeric array type to explain the essential idea of building relations on compound types. We believe that the same principle can be easily extended to other compound types.

Let x and y be two input variables. We build relations between them considering three situations: (1) x is a variable of numeric type and y a numeric array, (2) both x and y are numeric arrays with the same element type, and (3) both x and y are numeric arrays with different element types. The relations are shown in Tables 4–6, respectively. For brevity, we name the relations in Table 4 *ComRelations1*, the relations in Table 5 *ComRelations2*, and the relations in Table 6 *ComRelation3*.

In Table 4, $\mathit{NumericRelations}(x, y.\mathit{length})$ means to apply all the numeric relations given in Table 3 to x and $y.\mathit{length}$, where $y.\mathit{length}$ denotes the length of array y. The result of the operation $\mathit{elems}(y)$ represents all the elements of array y, thus, $\mathit{elems}(y) = \{x\}$ means that array y contains only x as its elements.

In Table 5, $\mathit{ComRelations}1(x.\mathit{length}, y)$ means that all the relations in Table 4 applies to variables $x.\mathit{length}$ and y; $\mathit{Min}(x.\mathit{length}, y.\mathit{length})$ denotes the smaller length of arrays

Table 4
Relations between an numeric variable and a numeric array

Basic relations	Complex relations	Boundary relations
$\mathit{NumericRelations}\ (x, y.\mathit{length})$	$x \in y$	$y.\mathit{length} = 0$
$\mathit{NumericRelations}(x, y[i])$, where $i = 0, \ldots, y.\mathit{length} - 1$	$x \notin y$	$\mathit{elems}(y) = \{0\}$
	$\mathit{elems}(y) = \{x\}$	

Table 5
Relations between two numeric arrays with the same element type

Basic relations	Complex relations	Boundary relations
$\mathit{ComRelations}1(x.\mathit{length}, y)$	$\mathit{elems}(x) \subseteq \mathit{elems}(y)$	$x = y \wedge x.\mathit{length} = 0$
$\mathit{ComRelations}1(y.\mathit{length}, x)$	$\mathit{elems}(y) \subseteq \mathit{elems}(x)$	
	$\mathit{elems}(x) \cap \mathit{elems}(y) = \{\ \}$	$x \neq y \wedge x.\mathit{length} = 0$
	$\mathit{elems}(x) \cap \mathit{elems}(y) \neq \{\ \}$	
	$x[i]\mathit{op}\ y[i]$, where $\mathit{op} \in \{>,<,=\}$, $i = \{1, \ldots, \mathit{Min}(x.\mathit{length}, y.\mathit{length})\}$	$x \neq y \wedge y.\mathit{length} = 0$

Table 6
Relations between two numeric arrays with different element types

Basic relations	Complex relations	Boundary relations
$\mathit{ComRelations}1(x.\mathit{length}, y.\mathit{length})$	$\mathit{elems}(x) \cap \mathit{elems}(y) = \{\ \}$	$x = y \wedge x.\mathit{length} = 0$
$\mathit{ComRelations}1(y.\mathit{length}, x.\mathit{length})$		$x \neq y \wedge x.\mathit{length} = 0$
		$x \neq y \wedge y.\mathit{length} = 0$

x and y. The basic relations are built between the lengths of arrays x and y; the complex relations are designed to define relations between their elements; and the boundary relations describe the boundary cases of the arrays.

In Table 6, since the elements of the two arrays x and y are of different type, not many complex relations between the elements of the two arrays can be built. In fact, comparisons between the values of incompatible types (e.g., $a > b$, where a is an integer and b a string) do not exist in programs that have passed the syntactical analysis by a compiler, therefore there is no need to consider relations between those types.

3.4. Relations on characters and strings

In some programs like compilers, comparison between characters or strings may be included in conditions of conditional or iteration statements. We design the relations on characters in Table 7 and the relations on strings in Table 8, and expect that test cases generated based on those relations will affect the evaluation of the conditions of conditional and iteration statements in the program. In Table 7, both x and y are variables of the character type. The comparison between two characters is actually made between the internal representation code (e.g., Unicode, ASCII code). Note that in Table 7, we use $Caps(x) = Caps(y)$ to mean the case-insensitive comparison of the equality between x and y, *Num* to mean the list of all digits from 0 to 9, and *EnglishLetters* to mean the list of English letters.

In Table 8, both x and y are variables of the string type and $x.length$ and $y.length$ denote their length, respectively. The equality and inequality of two string variables mean they have the same string and different string, respectively. $x > y$ and $x < y$ are evaluated based on the conventional algorithm for comparing strings. For example, “Happy” > “happy”, “Happiness” > “Happy”.

3.5. Relations between objects

An object in object-oriented programming language (e.g., Java, C#) usually has attributes, represented by instance variables. Each attribute can be a value of basic type, character type or string type. It can also be a value of a compound type like array or class. An object can be used as an individual composite value (it may consist of many attributes), it can also be included in a compound data structure (e.g., array). As an example, we discuss three kinds of relations: (1) relations between objects of the same class, (2) relations between an array of objects and an object of the same class, and (3) an array of objects and a data item that is one of the attributes of an object contained in the array.

The relations between objects of the same class can be built based on the relations between the attributes of the objects which were discussed previously. We use $Rel(o_1, o_2)$ to denote all the relations between the two objects o_1 and o_2. It contains all the relations between any attribute of o_1 and any attribute of o_2. Since $Rel(o_1, o_2)$ does not include new features, we do not discuss it in detail for brevity.

The relations between an array a of objects and an object o of the same class may include the relations given in Table 9. There is no useful basic relation between a and o because logical expressions in conditional and iteration statements of a program rarely involve numeric comparisons between the numeric aspects of a and o (e.g.,

Table 9
Relations between an array of objects and an object

Basic relations	Complex relations	Boundary relations
	$o \in a \wedge a.length > 1$	$o \in a \wedge a.length = 1$
	$o \notin a \wedge a.length > 1$	$o \notin a \wedge a.length = 1$
	$elems(a) = \{o\}$	$a.length = 0$

Table 7
Relations on characters

Basic relations	Complex relations	Boundary relations
$x = y$	$x = y \wedge x \notin \{Num, EnglishLetters\}$	$x \in \{Num\} \wedge y \in \{Num\}$
$x > y$	$x = y \wedge;x \in \{Num, EnglishLetters\}$	
$x < y$	$x \neq y \wedge x \notin \{Num, EnglishLetters\}$	$x \in \{EnglishLetters\} \wedge y \in \{EnglishLetters\}$
$Caps(x) = Caps(y)$	$x \neq y \wedge y \notin \{Num, EnglishLetters\}$	
	$x \neq y \wedge x \in \{Num, EnglishLetters\}$	
	$x \neq y \wedge y \in \{Num, EnglishLetters\}$	

Table 8
Relations on strings

Basic relations	Complex relations	Boundary relations
$ComRelations1(x.length, y.length)$	$x = y$	$x = y \wedge x = ""$
$ComRelations1(x.length, y.length)$	$x \neq y$	$x = y \wedge y = ""$
	$x > y$	$x \neq y \wedge x = ""$
	$x < y$	$x \neq y \wedge y = ""$
	$CompareIgnoreCase(x, y) = true$	$x = y \wedge x.length = Maximum$
		$x = y \wedge y.length = Maximum$
		$x \neq y \wedge x.length = Maximum$
		$x \neq y \wedge y.length = Maximum$

Table 10
Relations between an array of objects and a data item

Basic relations	Complex relations	Boundary relations
	$\exists_{i:inds(a)}\exists_{d:elems(a[i])} \cdot x = d$	$a.length = 0$
	$\neg\exists_{i:inds(a)}\exists_{d:elems(a[i])} \cdot x = d$	$\exists_{i:inds(a)}\exists_{d:elems(a[i])} \cdot x = d \wedge a.length = 1$
		$\neg\exists_{i:inds(a)}\exists_{d:elems(a[i])} \cdot x = d \wedge a.length = 1$

a.length, the number of the attributes of *o*). In contrast, several complex relations are possible, such as $o \in a \wedge a.length > 1$, stating that *o* is a member of array *a* and *a* contains more than one objects, and $elems(a) = \{o\}$, indicating that *a* contains only object *o* (possibly with more than one copies). Furthermore, three boundary relations for testing boundary situations are also provided in the table.

Let *c* be a class; *x* be an instance variable of *c*; and *a* be an array of objects of class *c*. Then, some commonly used relations between *a* and *x* are given in Table 10. Similar to Table 9, there is no useful basic relation, but there are two complex relations and three boundary relations given. The two complex relations describe the situation where *x* is same as one of the attributes of an object contained in array *a* and the situation otherwise, respectively. The first boundary relation in the first row of the table defines a situation where array *a* is empty. Another two boundary relations describe the situations where *x* is contained and not contained in an object of array *a*, respectively, and *a* includes only one object.

4. Process of test case generation

Fig. 1 illustrates the process of test case generation using our method, which is composed of two steps. The first step is to generate test cases to cover all functional scenarios defined in the specification because the specification is supposed to define what to be done by the program. The second step is to generate additional test cases to meet the specified coverage in the program. If, however, the test set generated in the first step already meets the specified coverage in the program, the second step may be skipped.

In general, when the number of input variables of an operation under testing is large, the number of relations used may also be large and the combinations of the relations may become complex as well. To efficiently generate a test set using the relation-based method, one may use the precondition and guard conditions (like $A_{pre} \wedge C_i$ in the FSF of operation *A* above) as the goal and select appropriate relations and/or their combinations to generate test set satisfying the guard conditions. Thus, we can avoid applying many unnecessary relations for generating the test set. Moreover, the conventional technique of inserting probes (e.g., a printing statement) in the program can be used to report the specified coverage status during testing, which helps the tester make decisions on the termination of the current test.

5. An example

We use a simplified *Withdraw* operation in the Automated Teller Machine (ATM) specification (Liu, 2003) and its corresponding program as an example to illustrate how the relation-based method is applied. The ATM specification was written in the Structured Object-Oriented Formal Language (SOFL) (Liu, 2004). The input of the *Withdraw* operation includes three data items: customer *id*, *password*, and *amount* for withdrawal. The operation authenticates the customer's *id* and *password* against the corresponding data items in the customer's bank account. If all the checking conditions are met, the requested amount of money will be delivered; otherwise, an error

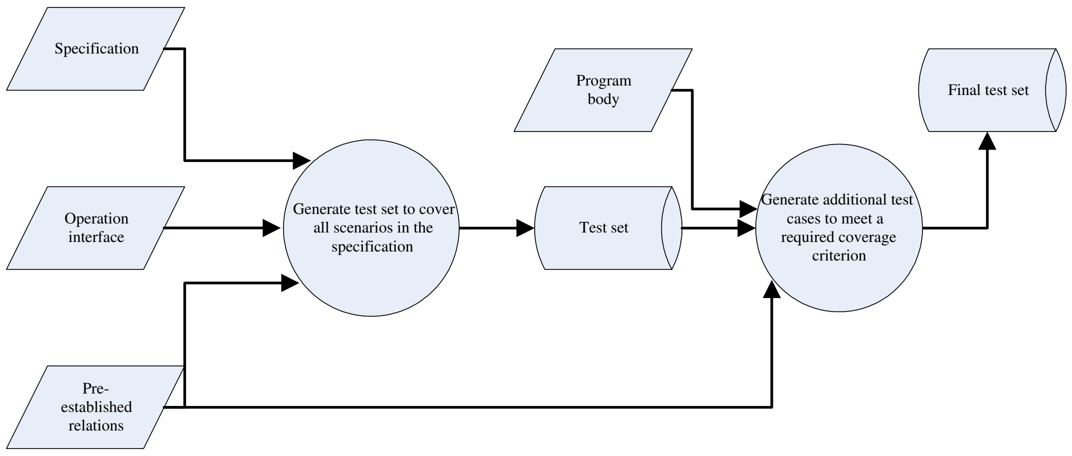
Specification
Operation interface
Pre-established relations
Generate test set to cover all scenarios in the specification
Test set
Program body
Generate additional test cases to meet a required coverage criterion
Final test set

Fig. 1. The test process using the relation-based method.

message will be issued. The specification of the operation is given as follows:

process *Withdraw*(*id*:*string*,*pass*:*int*,*amount*:*real*) *money*: *real*
ext *wr account_file*:*set of Account*;
wr err_message:*string*
pre *true*
post *let customer_account*: ∼*account_file*|
customer_account.acc_id = *id* ∧
customer_account.acc_password = *pass*
in
(*amount* ⩽ *customer_account.acc_amount_available* ∧
money = *amount* ∧
account_file = *union*(*diff*(∼*account_file*, *customer_account*),
modify(*customer_account*,
acc_amount_available → *customer_account.acc_amount_available* − *amount*))
∨
(*amount* > *customer_account.acc_amount_available* ∧
err_message = "The requested amount is not available"))
end_process

In this specification, **process** and **end_process** are two reserved words, indicating the start and the end of the specification, respectively; *id*,*pass*, and *amount* are all input variables; *money* is an output variable; and *account_file* and *err_message* are two external variables (state variables). *Account* is a composite type defined as follows:

Account = *composed of*
acc_id:*string*
acc_password:*int*
acc_balance:*real*
acc_amount_available:*real*
end

where *acc_balance* denotes the real balance of the account, while *acc_amount_available* represents the amount that can be withdrawn from the account currently. A type invariant on type *Account* is defined as follows:

$\forall_{x \in Account} \cdot x.acc\_amount\_available \leqslant x.acc\_balance,$

which states that the amount available for withdrawal currently is not greater than the balance.

Since the precondition of the operation *Withdraw* is true, its FSF can be obtained depending only on its postcondition:

(*customer_account*:
account_file|*customer_account.acc_id* = *id*∧
customer_account.acc_password = *pass*)∧
amount ⩽ *customer_account.acc_amount_available*)∧
(*money* = *amount*∧
account_file = *union*(*diff*(∼*account_file*,
customer_account),
modify(*customer_account*,
acc_amount_available-> *customer_account.acc_amount_available*-*amount*)))
∨
(*customer_account*:*account_file*|
customer_account.acc_id =
id ∧ *customer_account.acc_password* = *pass*)∧
amount > *customer_account.acc_amount_available*)∧
(*err_message* = "The requested amount is not available"))

The specification of *Withdraw* describes two functional scenarios: when the *amount* is within the amount available for withdrawal from the account and when the amount is beyond the withdrawal amount available. In the former, the requested amount will be delivered through the output variable *money*, while in the later the error message "The requested amount is not available" will be issued.

A program in C# implementing the operation *Withdraw* is given in Fig. 2. For the sake of space, we only show the relevant parts of the whole class *ATM*. Let us focus on the method *Withdraw* in the program that is expected to implement the corresponding specification. In addition to the input parameters *id*, *pass*, and *amount*, the method also uses the instance variables (corresponding to the external variables in the specification), such as *account_file*, *err_message*1, and *err_message*2. Compared to the specification, the program implements one more situation to check the existence of the customer account in the *account_file*. If it does, the processing of the withdrawal operation proceeds; otherwise, the error message "*The id or password is incorrect*." will be issued through the variable *err_message*2. This checking and the corresponding functional scenario are however not explicitly defined in the specification. It was actually missed in the specification, but recognized by the programmer to be necessary during the implementation.

Applying our relation-based method, we first select minimum relations introduced before to generate a test set, aiming to cover all the functional scenarios defined in the specification. The test cases and the expected results are given in Table 11. We generate the test cases by focusing on the relations between every two input variables (including input parameters, instance variables, and global variables used in the operation under test) each time. For example, the first row of the table shows a test case generated by applying the third boundary relation in Table 10 (i.e., $\neg\exists_{i:inds(a)}\exists_{d:elems(a[i])} \cdot x = d \wedge a.length = 1$) between the array type *account_file* and the parameter *id*. The second row of the table gives a test case generated by applying the first complex relation (i.e., $\exists_{i:inds(a)}\exists_{d:elems(a[i])} \cdot x = d$) in Table 10 between *account_file* and the parameter *pass*. The third row shows a test case that is also generated based on

```
class ATM
  {
      struct Account
      {
          String acc_id;
          intacc_password;
          double acc_balance;
          double acc_amount_available;
      }
      Account[ ] account_file = new account_file[10000];
      Stringerr_message1;
      Stringerr_message2;

      publicdoubleWithdraw(Stringid,intpass,doubleamount)
      {
          int i=0;
          bool found =true;
          Account customer_account=new Account( );
          double money=0;


          while (i <account_file.length && found) do
          {if (account_file[i].acc_id== id&& account_file[i].acc_password == pass)
              if(amount<=account_file[i].acc_amount_available)
               {money = amount;
                customer_account = account_ file[i];
                customer_account.acc_amount_available=customer_account.acc_amount_available-amount;
               account_file[i] =customer_account;
                found = false;
               return money;}
              else {err_message1= "The requested amount is not available.";
                    found = false;}
          else i++;

          if (found)
          {err_message2="The id or password is incorrect.";}

        }
    }
  }
```

Fig. 2. A program implementing the specification of the operation Withdraw.

Table 11
Test cases and expected results

account_file	*id*	*pass*	*amount*	*expected results*
{("SNishmura", 3941, 30000, 200)}	"KNakamura"	3219	20	"The id or password is incorrect."
{("SNishmura", 3941, 30000, 200), ("KNakamura", 3219, 40000, 200)}	"KNakamura"	3219	50	50
{("SNishmura", 3941, 30000, 200), ("KNakamura", 3219, 40000, 200)}	"SNishmura"	3941	300	"The requested amount is not available."
{("SNishmura", 3941, 30, 200), ("SMatunami", 3219, 40000, 200)}	"SNishmura"	3941	50	50

the first complex relation (i.e., $\neg\exists_{i:inds(a)}\exists_{d:elems(a[i])} \cdot x = d$) in Table 10 between *account_file* and the parameter *amount*. The fourth row of the table gives a test case generated by applying the first complex relation in Table 10 between *account_file* and *id*. The expected result for each test case is derived from the specification if the test case covers a functional scenario in the specification or from the program otherwise. In the former the expected result is formed based on the definition of the functional scenario in the specification, while in the latter the expected result is formed based on the structure of the relevant program path. Examining the postcondition of the operation *Withdraw*, we find that the four test cases in Table 11 cover all the functional scenarios in the specification. Furthermore, by executing the method *Withdraw* with the test cases respectively, we also find that they cover all the statements and branches in the body of the method. We choose to use the statement and branch coverage (*SB-coverage*) as the specified coverage in the program, therefore we do not need to generate additional test cases.

It is interesting to mention that the last test case in Table 11 actually helped us detect two errors in the program and its specification. The first error is that the program does not implement the type invariant on type *Account* to ensure that the withdrawal amount available currently is always less than or equal to the balance of the account. To deal with this situation, the program needs to check whether the amount is greater than the account balance before a withdrawal operation. If yes, no withdrawal is allowed and an appropriate error message should be delivered. The second error is that neither the specification defines the situation where the account balance is updated properly after a withdrawal, nor the program implements this function.

6. An experiment

To investigate the effectiveness of the relation-based method and identify points for further improvement, we took a software tool as the target system for an experiment. The tool supports the review (and analysis) of structural consistency between a program and its formal specification by assisting reviewers to check whether the program and its specification satisfy certain predefined criteria (Chen et al., 2005). The tool was constructed in Java under the Eclipse environment, and it contains 87 classes and 17380 lines of code. The major functions supported by the tool include *views derivation*, *checklist generation*, *review assistance*, and *review report generation*.

We selected 32 classes from the program for the experiment. These classes constitute the core components of the tool and therefore their quality is important for the whole tool system. The 32 classes, which contain 94 methods apart from their constructors, were divided into two groups known as *Group 1* and *Group 2*. *Group 1* contains a collection of classes that do not have complete specifications. *Group 2* contains a set of classes that inherit from the abstract class *ReviewDefect* contained in *Group 1*, and each class is designated to implement two methods *public String toString*( ) and *public String getComponent*( ). An important characteristic of classes in *Group 2* is that they contain methods whose body includes a number of guard conditions to check the equality between two inputs or between their attributes (e.g., $if(node1.type == node2.type)\{\ldots\}else\{\ldots\}$; $if(error.type! = 3)\{\ldots\}else\{\ldots\}$). Such classes and methods allow us to investigate the performance of our test method when it is used to test programs containing many equality comparisons in conditional and iteration statements.

The test of the system was conducted by independent testers, with the cooperation of the programmer who constructed the programs. The testers focused on the generation of test cases with our test method and run the programs with the test cases, while the programmer helped to determine errors. The test was carried out manually and took approximately 100 h to complete. The process of the testing includes the following four steps:

1. Identifying the inputs. The testers first tried to identify the inputs of programs to be tested by analyzing the relevant variables declared in the program. These variables include the input parameters and the referred global variables. The input parameters are directly declared in the method signature, while the referred global variables, which can be a field of the current class, the super class, an interface, a static field of some irrelevant class, or the object itself (e.g., **this**), can only be identified by the testers through reading the program code.
2. Selecting relations. We selected appropriate relations between the identified inputs for test case generation by considering two situations. The first situation is that the method to be tested has more than one input. In this case, we selected all the relations between every two of the inputs as the relations for test case generation. For example, assuming a Java method with the following signature

   $public\ static\ int\ get\_max(int\ x, int\ y, int\ z)$

   receives three integers x, y, and z as the inputs, and yield their maximum as the output, the relations we selected for test case generation is expressed as $Rel(x,y) \cup Rel(y,z) \cup Rel(x,z)$, the union of all the relations between x and y, all the relations between y and z, and all the relations between x and z.
   Another situation is that the method has only one input. In this case, we introduce an additional variable of the same type as the existing variable, and then select relations between the two variables. Although the additional variable is not used in the body of the method, it can help us select relations for test case generation. For example, suppose a Java method with the following signature

   $public\ static\ int\ length(int\ x1)$

   takes an integer $x1$ as the input and returns its length as the output. To obtain the relations for generating test cases to the method, we introduce a variable $x2$ that is of the same type as that of $x1$. We then apply the relations between two integers $x1$ and $x2$ given in Table 3 to generate test cases. Since a guard condition in the program is likely to involve $x1$ directly (e.g., $x1 > 10$) or indirectly through local variables (e.g., $y > 11$, where y is a local variable of the method and was assigned the result of the expression $x1 + 1$ previously), the change of the value for $x1$ in the test cases generated based on the relations between $x1$ and $x2$ will likely affect the evaluation results of the condition ($x1 > 10$ or $y > 11$).
3. Generating test cases using the relations. The testers generated test cases by first using simple relations and boundary relations. If the test cases did not cover all statements and branches in the program, the testers applied other basic, boundary, or complex relations to generate more test cases until the above condition is satisfied.

4. Executing the program with test cases. The testers executed each method with the test cases generated, and analyzed test results by checking the consistency between the actual results and the expected results. When the actual results are inconsistent with the corresponding expected results, it indicates the possibility of the program containing bugs.

Furthermore, the testers also investigated the performance in SB-coverage of the test cases generated by counting the traversed statement lines on each branch after each execution. The statistics of the test results for classes in *Group 1* are shown in Table 12. The item *Class name* indicates the name of the class tested; *Methods* denotes the number of methods defined in the corresponding class for testing; *Inputs* shows the number of input variables used; *Relation* shows the number of relations used for test case generation; *All lines* denotes the total number of statement lines of the methods in the class tested; *Uncovered lines* and *Covered lines* denote the numbers of lines of code that are not traversed and traversed, respectively, by the test cases generated; and the item *Covs* (abbreviation for *Coverages*) denotes the effectiveness of the corresponding test cases, which is calculated using the following formula:

$$\text{Coverage} = \frac{\text{Number of covered lines}}{\text{Number of all lines}} \times 100\%$$

Table 13 shows the statistics of the test results for classes in *Group 2*. The average coverage is approximately 50.2%, which is lower than the average coverage 63.9% for classes in *Group 1*.

Table 12
The statistics of the test results of Group 1

Class name	Methods	Inputs	Relations	All lines	Uncovered lines	Covered lines	Covs (%)
Parameter	4	8	66	61	8	53	86.9
ICONs	4	8	480	34	0	34	100.0
ReviewDefect	4	6	385	102	38	64	62.7
ReviewReport	2	8	680	28	16	12	42.9
Reorganize_SOFL	1	3	39	57	19	38	66.7
Views_Derivation_SDV	15	20	366	354	171	183	51.7
Views_Derivation_Eclipse_Project	1	2	15	10	5	5	50.0
ExampleFileFilter	7	9	361	62	15	47	75.8
SharedAlgorithms	10	15	366	107	22	85	79.4
Total numbers	48	79	2758	815	294	521	63.9

Table 13
The statistics of the test results of Group 2

Class name	Methods	Inputs	Relations	All lines	Uncovered lines	Covered lines	Covs (%)
ReviewDefect1_1	2	3	15	28	10	18	64.3
ReviewDefect1_2	2	6	12	56	24	32	57.1
ReviewDefect2_1	2	3	15	28	10	18	64.3
ReviewDefect2_2	2	6	12	56	24	32	57.1
ReviewDefect2_3	2	4	15	50	19	31	62.0
ReviewDefect2_4	2	4	15	50	19	31	62.0
ReviewDefect2_5	2	8	25	56	24	32	57.1
ReviewDefect3_1	2	4	15	41	18	23	56.1
ReviewDefect3_2	2	4	15	40	18	22	55.0
ReviewDefect3_3	2	4	15	41	18	23	56.1
ReviewDefect3_4	2	6	12	57	27	30	52.6
ReviewDefect3_5	2	6	12	87	53	34	39.1
ReviewDefect3_6	2	6	12	67	38	29	43.3
ReviewDefect3_7	2	6	12	65	38	27	41.5
ReviewDefect3_8	2	6	12	67	38	29	43.3
ReviewDefect4_1	2	4	15	41	17	24	58.5
ReviewDefect4_2	2	4	15	41	17	24	58.5
ReviewDefect4_3	2	4	15	41	17	24	58.5
ReviewDefect4_4	2	6	12	56	27	29	51.8
ReviewDefect4_5	2	6	12	84	52	32	38.1
ReviewDefect4_6	2	6	12	73	42	31	42.5
ReviewDefect4_7	2	6	12	74	42	32	43.2
ReviewDefect4_8	2	6	12	75	42	33	44.0
Total numbers	46	118	319	1274	634	640	50.2

Table 14
An example of effective test using the relation-based method

Program:

```
public static String get_displayname(String icon_name){
      for(int i=0;i<icons.length;i=i+4){
         if(icon_name.compareToIgnoreCase(icons[i])==0){
            return icons[i+1];
         }
      }
      return "";
   }
```

Inputs: final static String[] icons, String icon_name
* icons[] has been declared as an constant.

Relations, test cases, and expected results:

Relations:	Input: *icon_name*	Expected Results
$icon\_name \in icons[]$	"Save CVIEW"	Error_message
$icon\_name \notin icons[]$	"No such an icon name"	Error_message
$icon\_name = icons[0]$	"Main_Window_ICON"	"Main Window ICON"
$icon\_name = icons[icons.len - 1]$	"/Resources/Delete16.gif"	Error_message
$icon\_name = icons[\lfloor(icons.len - 1)/2\rfloor]$	"Sub_Add_CONTROL"	"Add Control"
$icon\_name = icons[\lceil(icons.len - 1)/2\rceil]$	"/Resources/newprocess.gif"	Error_message
...	...	...

Uncovered codes: none

Apart from the assessment of the test method in SB-coverage, we have also detected 24 errors in the program. An example of the errors found is that the method *toString*( ) in Table 15 does not produce expected error message, and this error was detected using the test cases given in the same table.

The experiment has shown that our test method is more effective for some kinds of programs than some other kinds of programs in SB-coverage, as indicated by the test results in Tables 12 and 13. Although the average coverages of the two groups are not of great difference, almost all coverages of methods in *Group 2* are lower than the average coverage of methods in *Group 1*. To find the reason for the difference, we made an intuitive comparison between the method in *Group 1* with the coverage 100% and a method in *Group 2* with the coverage under 50%. Tables 14 and 15 show the test with 100% coverage and the test with less than 50% coverage, respectively. All the untraversed statements, if any, are indicated by writing them in italics font.

On the basis of our analysis, we have found two reasons for the low coverage of the methods in *Group 2*. One reason is that there are many guard conditions to check the equality between inputs or between an input and an constant (e.g., *switch(checked)*{*case 5:*...}). Since such a condition requires very specific values of the relevant inputs and/or constants to meet the condition, it is quite difficult for test cases to satisfy the condition by generating them based only on the pre-established relations. Another reason is that the inputs of a program are sometimes used in conditions independently. Thus, changing the relations between inputs would generate test cases with little impact on the evaluation of the conditions in the program.

In addition to the reasonably high coverage of our method, we believe that the major strength of the relation-based method is its capability for automatic test case generation. The reason is that all the relations are established in advance and selecting values from the related types to satisfy the relations is a straightforward process. With effective tool support, this feature will considerably benefit the testing process by enhancing the test efficiency.

On the other hand, the test method also faces several challenges when used in practice. The first challenge is that the pre-established relations offer few instructions on the derivation of expected test results. The expected results actually have to be worked out by the tester based on either specification or program or both, and such an activity is unlikely to be automatic. However, if the specification is written in formal notation and defines all the desired functions to be implemented, it may be possible to algorithmically derive effective test oracles from the specification (McDonald and Strooper, 1998; Offutt and Liu, 1999). The second challenge is that the test method suggests little for choosing the pre-established relations for test case generation. In our experiment, we first tried to apply the basic relations and the boundary relations, and then tried to apply the complex relations if the basic and boundary

Table 15
An example of ineffective test using the relation-based method

Program:

```
public String toString() {
      switch (checked) {
      case 1:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is implemented in the program!";
      case 2:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is not implemented in the program!";
      case 3:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is implemented in the program or not!";
      case 4:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is implemented in the program!";
      case 5:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is not implemented in the program!";
      case 6:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is implemented in the program or not!";
      default:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV!";
      }
   }
```

Inputs: int checked, int node.id
*node is a global variable

Relations and test cases:

Relations	Input: *checked*	Input: *node.id*	Expected Results
$checked = node.id$	20	20	Error_Message
$checked = node.id + 1$	21	20	Error_Message
$checked = node.id + 10$	30	20	Error_Message
$checked = node.id + 100$	120	20	Error_Message
$checked = node.id - 1$	19	20	Error_Message
$checked = node.id - 10$	10	20	Error_Message
$checked = node.id - 100$	-80	20	Error_Message
$checked > 0 \wedge checked = node.id + 1$	21	20	Error_Message
$checked = 0 \wedge checked = node.id + 1$	0	-1	Error_Message
$checked < 0 \wedge checked = node.id + 1$	-19	-20	Error_Message
...	...	...	...

Uncovered code:

```
public String toString() {
      switch (checked) {
      case 1:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is implemented in the program!";
      case 2:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is not implemented in the program!";
      case 3:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is implemented in the program or not!";
      case 4:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is implemented in the program!";
      case 5:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is not implemented in the program!";
      case 6:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV is implemented in the program or not!";
      default:
         return get_prefix_string() + "the Class/Type Node: " + node.id  + " in the SSV!";
      }
   }
```

relations are insufficient for the expected coverages, but which basic and boundary relations should first be applied and which complex relations should then be used for test case generation still needs tester's judgments. The third challenge is the lack of an effective tool support for the method. Although the method can be applied to generate test cases under existing tool support, such as *JUnit*, the tester still needs to select, apply, and manage relations manually. Our experience in the experiment indicates that it would be inefficient for testing large-scale and complex software systems without powerful tool support. We will continue to make efforts to address the challenges above in the future.

7. Related work

Formal specification-based testing has been extensively researched over the last twenty years, but they all face the challenges discussed in Section 1. Ammann and Offutt (1994) described a way to apply category-partition testing based on formal specifications. They proposed to construct test specifications from functional specifications and defined a minimal criterion for generating test cases using category-partition approach based on test specifications. The assumption of this work is that the functional specification used for testing is valid and complete in defining expected behaviors. Stocks and Carrington (1996) used formal methods to describe test artifacts, specifically test frames (sets of test inputs that satisfy a common property) and test heuristics (specific methods for testing software). Blackburn et al. (2005) proposed an interface-driven approach that combines requirement modeling to support automated test case and test driver generation. It focuses on how test engineers can develop more reusable models by clarifying textual requirements as models in terms of component or system interfaces. Barnett et al. (2003) presented an approach to modeling use cases and scenarios in the Abstract State Machine Language (AsmL) and discussed how to generate test cases based on scenarios for validation and verification purposes. Chen et al. (2004) have suggested a similar way to generate test cases for integration testing based on scenarios, but the scenarios are derived from data flow specifications rather than from abstract state machines.

While the work above focuses on technical contributions to specification-based testing, Dalal et al. (1999) have made an effort to investigate the situation of specification-based testing in practice. Their study suggests that it is natural and straightforward for testers to take the activity of specifying a software component's inputs and that specification-based testing has helped testers reveal numerous defects. In the meantime, however, they have also found that specification-based testing may be challenging when test engineers face programs without properly updated specifications and when they are under severe pressure by the short release cycles expected in today's software markets. Our relation-based test method described in this paper addresses those challenges because test case generation using our method is unlikely affected by the gap between specifications and their corresponding programs, and the generation can be automated.

8. Conclusion

We have described a relation-based method for testing programs. The core idea of the method is to define a set of relations between variables of various types in advance, and then those relations are applied selectively to the input variables of the program under testing to generate test cases. Appropriate test cases need to be produced to cover all functional scenarios defined in the specification, and all statements in the program. We have shown how the test method can be applied with an example, and investigated its effectiveness by conducting an experiment on testing a software tool system. The result of the experiment shows that the test method is relatively effective for test case generation in terms of SB-coverage.

As we have discussed in the last paragraph of Section 6, our test method also faces three challenges in the selection of appropriate relations and tool support when used in practice. We will continue to work on those issues to improve the method and to provide effective tool support in the future.

Acknowledgements

We thank all the students for joining the experiment, the Ministry of Education, Culture, Sports, Science and Technology of Japan for its financial support under Grant-in-Aid for Scientific Research (No. 18500027), and Hosei University for a special research grant in 2006 to support this research.

References

Ammann, P., Offutt, A.J., 1994. Using formal methods to derive test frames in category-partition testing. Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94), June 1994. IEE Computer Society Press, Gaithersburg MD, pp. 69–80.

Barnett, M., Grieskamp, W., Gurevich, Y., Schulte, W., Tillmann, N., Veanes, M., 2003. Model-based testing. Proceedings of Second International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM2003), May 2003. In: LNCS. Springer-Verlag.

Beizer, B., 1990. Software Testing Techniques, second ed. International Thomson Computer Press.

Beizer, B., 1995. Black-Box Testing. John Wiley and Sons, Inc.

Bernot, G., Gaudel, M.C., Marre, B., 1991. Software testing based on formal specifications: a theory and a tool. Software Engineering Journal 6 (6), 387–405.

Blackburn, M.R., Busser, R.D., Nauman, A.M., 2002. Interface-driven, model-based test automation. In: Proceedings of 2005 International Conference on Software Testing, Analysis and Review. Software Quality Engineering, November 14–18, 2002, pp. 27–30.

Chen, Y., Liu, S., Nagoya, F., 2004. An approach to integration testing based on data flow specifications. Proceedings of First International Colloquium on Theoretical Aspects of Computing, September 20–24 2004. In: LNCS. Springer-Verlag, Guiyang, China, pp. 235–249.

Chen, Y., Liu, S., Nagoya, F., 2005. A review approach to detecting structural consistency violations in programs. In: Yang, K.K., Akingbehin, K. (Eds.), Proceedings of Fourth ACIS International Conference on Computer and Information Science (ICIS'05), July 14–16 2005. IEEE Computer Society Press, Jeju Island, Korea, pp. 61–66.

Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M., 1999. Model-based testing in practice. Proceedings of the 21st International Conference on Software Engineering. IEEE Computer Society Press, pp. 285–294.

El-Far, I.K., Whittaker, J.A., 2001. Model-based Software Testing. Wiley.

Kaner Cem, 2000. Grey box testing. In: Kaner Cem, Lawrence Brian (Eds.), Los Altos Workshop on Software Testing #9. Sunnyvale, CA, March 2000.

Larman, C., Basili, V.R., 2003. Iterative and incremental development: a brief history. The Computer, 47–56.

Liu, S., 2003. A case study of modeling an atm using SOFL. Technical Report HCIS-2003-01, CIS, Hosei University, Koganei-shi, Tokyo, Japan.

Liu, S., 2004. Formal Engineering for Industrial Software Development Using the SOFL Method. Springer-Verlag, ISBN 3-540-20602-7.

Liu, S., Nagoya, F., Chen, Y., Goya, M., McDermid, J.A., 2005. An automated approach to specification-based program inspection. In: Lau, K.K., Banach, R. (Eds.), Proceedings of the Seventh International Conference on Formal Engineering Methods (ICFEM2005), November 1–4 2005, LNCS, 3785. Springer-Verlag, Manchester, UK, pp. 421–434.

McDonald, J., Strooper, P., 1998. Translating object-Z specifications to passive test oracles. In: Staples, J., Hinchey, M.G., Liu, S. (Eds.), Proceedings of the Second International Conference on Formal Engineering Methods (ICFEM98), December 1998. IEEE Computer Society Press, Brisbane, Australia, pp. 165–174.

Offutt, A.J., Liu, S., 1999. Generating test data from SOFL specifications. Journal of Systems and Software 49 (1), 49–62.

Shannon, M., Miller, G., Prewitt, R.J., Loveland, S., 2004. Software Testing Techniques: Finding the Defects that Matter. Programming Series, Charles Biver Media, 2004.

Stocks, P., Carrington, D., 1996. A framework for specification-based testing. IEEE Transactions on Software Engineering 22 (11), 777–793.

Woodcock, J., Davies, J., 1996. Using Z: Specification, Refinement, and Proof. Prentice Hall.

Shaoying Liu is a Professor of Software Engineering at Hosei University. He holds a Ph.D from the University of Manchester, U.K. His research interests include Formal Engineering Methods, Software Development Methodology, Software Inspection, Software Testing, Dependable Complex Computer Systems, and Intelligent Software Engineering Environments. He has published over 80 academic papers in refereed journals and international conferences, and recently published a book titled "Formal Engineering for Industrial Software Development Using the SOFL Method" with Springer-Verlag. He has served as general chairs, program chairs, or program committee members for numerous international conferences, and is currently serving on the editorial board for the Journal of Testing, Verification and Reliability.

He is a member of IEEE Computer Society, British Computer Society, and Japan Society for Software Science and Technology.

Yuting Chen holds a Ph.D in Computer Science from Hosei University, Japan. He is currently a research fellow at the University of Texas at Dallas. His research interests include software inspection, specification-based testing, formal engineering methods, and software support environments.