A New Software Testing Approach Based on Domain Analysis of Specifications and Programs

Ruilian Zhao
Computer Science Dept.
Beijing University of
Chemical Technology

Michael R. Lyu
Computer Science Dept.
Chinese University of
Hong Kong

Yinghua Min
Institute of Computing Tech.
Chinese Academy of Sciences
in Beijing

Abstract

Partition testing is a well-known software testing technique. This paper shows that partition testing strategies are relatively ineffective in detecting faults related to small shifts in input domain boundary. We present an innovative software testing approach based on input domain analysis of specifications and programs, and propose the principle and procedure of boundary test case selection in functional domain and operational domain. The differences of the two domains are examined by analyzing the set of their boundary test cases. To automatically determine the operational domain of a program, the ADSOD system is prototyped. The system supports not only the determination of input domain of integer and real data types, but also non-numeric data types such as characters and enumerated types. It consists of several modules in finding illegal values of input variables with respect to specific expressions. We apply the new testing approach to some example studies. A preliminary evaluation on fault detection effectiveness and code coverage illustrates that the approach is highly effective in detecting faults due to small shifts in the input domain boundary, and is more economical in test case generation than the partition testing strategies.

1. Introduction

With the expansion of software system size and complexity, there is an ever-increasing demand for innovative testing schemes for software quality and reliability. Software testing can usually be classified into two categories: functional testing and structural testing depending on whether it is necessary to analyze and execute program source codes. Structural testing strategies make use of program control structures to generate test cases [1,2,3]. In functional testing, the only information used to develop test cases is software specifications [1,4,5]. The implementation details are ignored. Hence, some shortcomings result from the dependence on program control structures or software specifications. An obvious problem is that program control structures or software specifications themselves may be incorrect. This makes it difficult to detect specification faults that are not reflected in program structures and program faults that have nothing to do with software specifications. Consequently, if testers have some knowledge about the structure as well as the specification of the program under test, better test effectiveness can be achieved.

To test a program, it is necessary to select test data from the program input domain. As it is usually too large to be exhaustively exercised, the usual way for testing is to select a relatively small subset to represent. Therefore, a key issue in software testing is how to select test data from program input domain to detect as many faults as possible with a minimum cost.

There are a large number of test data selection strategies, such as equivalence partitioning [6,7], boundary value analysis [7], path testing [1,3], domain testing [1,8] and so on. All of these strategies are based on partitioning input domain, referred to as partition testing. The input domain is divided into some sub-domains, and one or more representatives from each sub-domain are selected to test the program. However, in this paper, we will show with an example that partition testing strategies are relatively ineffective in detecting faults having to do with small shifts in input domain boundary.

This paper presents a new software testing approach based on input domain analysis of specifications as well as programs. As discussed in [9], a system is defined by functions in the requirement phase, and is described by operations in the development phase. Software specification defines an input domain termed “functional domain,” while the code implementation also specifies a domain termed “operational domain.” If the two domains are not coincided with each other exactly, some software faults may be located.

Considering those cases near domain boundary to be more sensitive to software faults than others, this paper

Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE’03)
1071-9458/03 $ 17.00 © 2003 IEEE

proposes the principle and procedure of boundary test case selection, and designs a set of boundary test cases of functional domain as well as that of operational domain. The coincidence of the two domains is examined by analyzing the two sets. If they are not equal to each other, there are some discrepancies between the specification description and the code implementation. Some software faults can thus be detected, and either the specification or the program, or both, should be repaired. In order to obtain operational domain of a program, we have developed an automated determination system of operational domain, called *ADSOD*, which supports not only the determination of input domain of integer and real data types, but also non-numeric data types such as characters and enumerated types. It consists of several modules in finding illegal values of input variables with respect to special expressions. As a result, the domain of the input variable can be determined. We apply the new testing approach to some example studies. A preliminary evaluation on fault detection effectiveness and code coverage indicates that the testing approach is effective in detecting faults related to small shifts in the domain boundary with little cost overhead.

The remainder of this paper is organized as follows. Section 2 reviews some partition testing strategies. Section 3 presents an innovative software testing approach based on input domain analysis of specifications and programs, proposes the principle of boundary test case selection, and outlines the process of test case selection. Section 4 introduces an automated determination system of operational domain *ADSOD,* and describes a module finding illegal values of input variables for specific expressions. Section 5 compares the effectiveness of the testing approach with partition testing strategies in fault detection by an example. Finally, conclusion is provided in Section 6.

2. Partition testing strategies

Partition testing is a well-known software testing technique. All partition testing strategies are based on partitioning the input domain of the program under test. By dividing a program input domain into some disjoint or non-disjoint sub-domains, one or more representatives from each sub-domain are selected to test the program [10].

Path testing and domain testing are two typical strategies of partition testing. In the following, we briefly discuss the two testing strategies.

Path testing requires that each path in tested program be executed by at least one test case. However, it is impractical since there may exist a huge (or even an infinitive) number of different paths in a program with loops. An alternative is to test some representative paths. Hence, we make use of boundary-interior path testing instead of path testing to check the program. Boundary-interior path testing is a restricted version of path testing in which the number of test cases is limited by grouping paths and then testing a few representative paths from each group. It lies between path and All-DU-path testing in the subsume orderings of structural testing strategies [3], as shown in Fig.1.

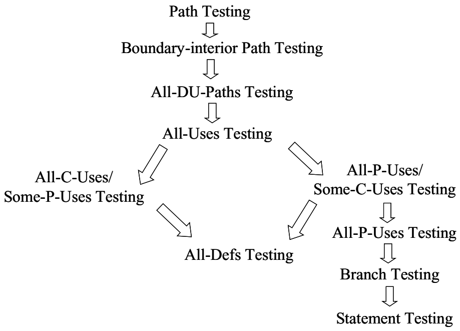

Fig. 1. Partial ordering of some structural testing strategies

Boundary-interior path testing deals with two classes of paths with respect to each loop from each group of similar paths that differ only in the number of times that they iterate loops. The first class paths enter into the loop but do not iterate it while the second class paths iterate the loop at least once. The former is called as boundary test in which different paths inside the loop are executed. The latter is interior test in which different paths through the first iteration of the loop are executed. For example, in a *FOR* loop that contains a single *IF-THEN-ELSE* statement, there are two boundary tests for this loop, one for each branch of the *IF-THEN-ELSE* statement, and both will exit the loop immediately. There are four interior tests for the loop, each of which will execute the body of the loop a second time, corresponding to the four permutations of the branches of the *IF-THEN-ELSE*, i.e., True-True, True-False, False-True, False-False. After the second execution of the body of the loop, each interior test path may exit the loop or iterate it any additional number of times, taking either one of the branches in the *IF-THEN-ELSE* statement. If a program does not contain any loops, boundary-interior path testing is equivalent to path testing.

Each path has a path domain, which is the set of all inputs that cause the path to be traversed during the program execution. Therefore, the input domain of a program may be partitioned into some sub-domains by its all boundary-interior paths, and then test cases corresponding to each sub-domain are developed to exercise the paths.

Domain testing is effective in identifying border shift faults of a path domain. A path domain is surrounded by a boundary, and the segments of the boundary are called borders. The simplified domain testing strategy requires two types of points to be selected as test cases to detect border shift involved in a chosen path [8]. One is *ON* test point, and the other is *OFF* test point. The *ON* test point can be anywhere on the given border, but it must satisfy the path condition associated with the border. The *OFF* test point should be as close to the *ON* test point as possible, and lies outside the border. The only way that a border shift can escape fault detection is if the correct border passes between the *ON* and *OFF* test points. By selecting an *ON-OFF* pair very close to each other, this is unlikely to happen.

A main shortcoming of partition testing strategies is that they provide no guidelines for selecting test data from a path domain and many errors along a path can be found only if the path is executed with values from a small subset of its domain. Moreover, these testing strategies require more test cases than the testing approach that we will introduce in this paper.

3. The new testing approach

In this section, we present a new software testing approach, referred to as a domain analysis testing approach based on specifications and programs. Test cases are generated from coincidence verification of functional domain and operational domain of the program under test. In what follows, we will introduce in detail the principle and procedure of boundary test case selection in functional domain and operational domain.

3.1 The principle of boundary test case selection

Suppose that a program fulfills function *F*,

$$F: \quad (x_1, x_2, \cdots, x_n \rightarrow y_1, y_2, \cdots, y_m \quad)$$

where $x_i \ (i = 1,2,\cdots,n)$ is an input variable and $y_j \ (j = 1,2,\cdots,m)$ is an output variable of the program. The *domain* D_{x_i} of input variable x_i is a set of all values that x_i can hold. By the *domain D* of a program, we mean a cross product $D = Dx_1 \times Dx_2 \times \cdots \times Dx_n$ [11]. Let $I = (x_1, x_2, \cdots, x_n)$ be a vector of input variables, and *E* denotes a space of *n*-dimension vectors, here we call it *input space*. Then the domain *D* of a program can be thought of a subset of input space *E*.

Let $B(D)$ denote the set of boundary points of domain *D*. More precisely, $B(D)$ is the set of point *p* such that in any small neighborhood of *p*, there are some points in domain *D*, and others are out of domain *D*. The projection of domain *D* on x_i axis, represented by $D|x_i$, is a subset of the one dimension space X_i. It satisfies

$$D|x_i = \{x_i \in X_i | \exists (x_1^0, \cdots, x_{i-1}^0, x_i, x_{i+1}^0, \cdots, x_n^0) \in D\}$$

Definition 1: *Reflecting domain of projection.*

By reflecting domain $R(D, x_i = a)$ *of projection of domain D on x_i axis, we mean in n-1 dimension space*

$$R(D, x_i = a) = \{(x_1, \cdots, x_{i-1}, a, x_{i+1}, \cdots, x_n) \in D\}$$

where $a \in D|x_i$.

The following reflecting domains are defined in succession:

$$RD_0 = D$$

$$RD_1 = R(RD_0, x_{i_1} = a_{i_1}), \quad a_{i_1} \in RD_0|x_{i_1}$$

$$RD_2 = R(RD_1, x_{i_2} = a_{i_2}), \quad a_{i_2} \in RD_1|x_{i_2}$$

$$\cdots\cdots$$

$$RD_{n-1} = R(RD_{n-2,} x_{i_{n-1}} = a_{i_{n-1}}), \quad a_{i_{n-1}} \in RD_{n-2}|x_{i_{n-1}}$$

where $a_{i_1}, a_{i_2}, \cdots, a_{i_{n-1}}$ are constants. By the definition, $RD_j \ (0 \le j \le n-1)$ is an *n-j* dimension subset of the original domain *D*, and $RD_{n-1} \subseteq RD_{n-2} \subseteq \cdots \subseteq RD_0$.

Theorem 1: *Let D be an n-dimension bounded closed convex domain. Let* $RD_0, RD_1, \cdots, RD_{n-1}$ *be defined as above. Then* $B(RD_{n-1}) \subseteq B(RD_{n-2}) \subseteq \cdots \subseteq B(RD_0)$.

Proof. Since *D* is a bounded closed convex domain, we can prove recursively that $RD_0, RD_1, \cdots, RD_{n-1}$ are bounded closed convex domain. Thus $B(RD_j)$ is well defined and $RD_j \subseteq RD_{j-1}$ for all $j = 1, 2, \cdots, n-1$. Now, we prove $B(RD_j) \subseteq B(RD_{j-1})$. Suppose that $RD_j = R(RD_{j-1}, x_k = a_k)$ for some fixed $1 \le k \le n-1$, and point $q^* = (x_1^*, x_2^*, \cdots, x_{k-1}^*, x_{k+1}^*, \cdots, x_{n-j}^*) \in B(RD_j)$. It suffices to prove that point $q' = (x_1^*, x_2^*, \cdots, x_{k-1}^*, a_k, x_{k+1}^*, \cdots, x_{n-j}^*) \in B(RD_{j-1})$. Obviously, $q^{'} \in RD_{j-1}$. Assume that $N(q^{'})$ is an arbitrary neighborhood of $q^{'}$. Let $M = R(N(q^{'}), x_k = a_k)$. It is easy to see that *M* is a neighborhood of q^*. By the definition of boundary points, there exist points $q_1^* = (y_1, \cdots, y_{k-1}, y_{k+1}, \cdots, y_{n-j})$ and $q_2^* = (z_1, \cdots, z_{k-1}, z_{k+1}, \cdots, z_{n-j})$ such that $q_1^*, q_2^* \ne q^*$, $q_1^* \in RD_j$ and $q_2^* \notin RD_j$. We write $q_1^{'} = (y_1, \cdots, y_{k-1}, a_k, y_{k+1}, \cdots, y_{n-j})$ and $q_2^{'} = (z_1, \cdots, z_{k-1}, a_k, z_{k+1}, \cdots, z_{n-j})$. It follows that $q_1^{'} \in RD_{j-1}$ and $q_2^{'} \notin RD_{j-1}$. Thus, $q^{'} \in B（RD_{j-1}）$. We derive $B(RD_j) \subseteq B(RD_{j-1})$. Using the formula repeatedly, we have $B(RD_{n-1}) \subseteq B(RD_{n-2}) \subseteq \cdots \subseteq B(RD_0)$.

This can be explained with a sphere domain *D*, shown in Fig.2. The reflecting domain of projection of *D* on x_1

axis, i.e. $RD_1 = R(D, x_1 = a_1)$, is the gray circle, and that of RD_1 on *x2* axis, i.e. $RD_2 = R(RD_1, x_2 = a_2)$, is the line Q_1Q_2. It can be seen that the set of boundary points of the line belongs to that of the circle, and the set of boundary points of the circle belongs to that of the sphere, namely $B(RD_2) \subseteq B(RD_1) \subseteq B(D)$.

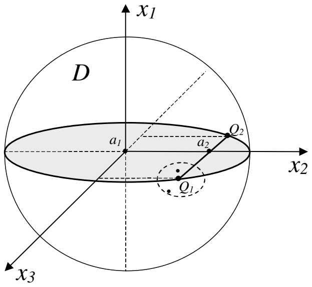

Fig.2 Reflecting domains of projection

By above successive definition, we have that $RD_{n-1} = R(RD_{n-2}, x_{i_{n-1}} = a_{i_{n-1}}) = \cdots = R(\cdots R(R(D, x_{i_1} = a_{i_1}), x_{i_2} = a_{i_2})\cdots, x_{i_{n-1}} = a_{i_{n-1}})$ is a one dimension subset of the original domain *D*, that is, $RD_{n-1} = \{(a_1, a_2, \cdots, x_{i_n}, \cdots a_n)\}$. We denote it by $RD(x_{i_n})$. Then, $B(RD(x_{i_n})) \subseteq B(D)$ and the number of boundary points of $RD(x_{i_n})$ is equal to 2.

Definition 2: *The set of boundary test cases T(D).*

Let a_{i_j} *be a center point of projection of domain* RD_{j-1} *on* x_{i_j} axis $(j = 1, 2, \cdots, n-1)$, *respectively. The set of boundary test cases of domain D is defined as follows*:

$$T(D) = \{(t_1, t_2, \cdots, t_n) \mid (t_1, t_2, \cdots, t_n) \in B(RD(x_1)) \cup B(RD(x_2)) \cup \cdots \cup B(RD(x_n))\}$$

As mentioned above, software specification defines a functional domain, denoted by D_f, while code implementation specifies an operational domain, represented by D_p. If the two domains do not precisely coincide with each other, some software faults can be located. The following claim describes how to select boundary test cases to verify the coincidence of the two domains.

Claim: *Suppose functional domain* D_f *and operational domain* D_p *are bounded closed convex. Then, the sufficient and necessary condition that domain* D_f *coincides with domain* D_p *is:*

1). $T(D_f) = T(D_p)$;

2). *For point* q^* *in any neighborhood of point q,* $q \in T(D_f)$ *or* $q \in T(D_p)$, *then, either* $q^* \in D_f \cap D_p$ *or* $q^* \notin (D_f \cup D_p)$.

The necessity is obvious. We demonstrate the sufficiency by contradiction. Suppose that condition 1) holds but $D_f \neq D_p$. Then, there must be a point q', such that $q' \in D_f$ or $q' \in D_p$, but $q' \notin D_f \cap D_p$. Let point q'' be a boundary point of domain D_f and D_p, i.e. $q'' \in T(D_f) \cap T(D_p)$, close to q', then, there is at least one point q^* on the shortest path between q' and q'' that does not satisfy condition 2), as shown in Fig.3.

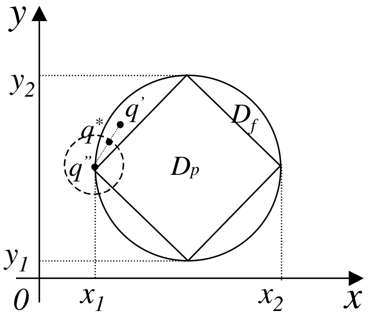

Fig.3 Input domain D_f and D_p

For example, suppose that domain D_f is a circle, while domain D_p is a square inside in two-dimension space in Fig.3. The projections of domain D_f and domain D_p are [x_1, x_2] on the *x*-axis and [y_1, y_2] on the *y*-axis, respectively. Four boundary points are $(x_1, \frac{y_1 + y_2}{2})$, $(x_2, \frac{y_1 + y_2}{2})$, $(\frac{x_1 + x_2}{2}, y_1)$, $(\frac{x_1 + x_2}{2}, y_2)$, respectively, so $T(D_f) = T(D_p)$. But in the neighborhood of boundary point q'', there exist at least a point q^* in D_f and not in D_p. Thus, $D_f \neq D_p$.

3.2 Test data generation based on the domain analysis

Input space *E* can be partitioned into four subspaces, that is, $E = E_1 \cup E_2 \cup E_3 \cup E_4$,

where $E_1 = \{x \mid x \in D_f \cap D_p\}$

$E_2 = \{x \mid x \in \overline{D}_f \cap D_p\}$

$$E_3 = \{x | x \in D_f \cap \overline{D}_p\}$$

$$E_4 = \{x | x \in \overline{D}_f \cap \overline{D}_p\}.$$

Subspace E_1 indicates that a specification and corresponding program have the same input domain, but it should be further verified whether the specification description coincides with the code implementation or not. Subspace E_2 displays that the program has produced a result that is not stated by the specification. In this instance, the specification should be supplemented, or the program should reduce its operational domain. Subspace E_3 shows that the program does not accomplish some requirements of the specification. In other words, the program should be modified to implement the requirements. Finally, subspace E_4 concerns with some exceptions, which are neither required by the specification nor involved in the program. Here the program behaviors are unpredictable for unspecified inputs. In high reliable software, exception handling is indispensable since, usually, subsystems are imbedded in a large system. If unspecified inputs occasionally occur, the system may produce fatal failure. Therefore, the processes must be inspected not only in the subspace E_1, but also in E_2, E_3 and E_4.

As declared previously, if there are discrepancies between functional domain D_f and operational domain D_p, some software faults can thus be detected. In order to check the coincidence of domains D_f and D_p, not only the sets of boundary test cases $T(D_f)$ and $T(D_p)$ need to be examined, but also some test cases in neighborhoods of the boundary points should be taken into consideration by above claim.

The test data generation based on the domain analysis consists of the following steps.

1) Find out all input variables $(x_1, x_2, \cdots, x_n)$ from both the specification and the program under test. These input variables may be dependent on each other, i.e., there may be some correlation between them.

2) Derive $T(D_f) = T(D_p)$

That is, for all $i \quad (i = 1,2,\cdots,n)$

a): compute $B(RD_f(x_i))$ and $B(RD_p(x_i))$.

b): check whether $B(RD_f(x_i)) = B(RD_p(x_i))$ is true or false.

If $B(RD_f(x_i)) \neq B(RD_p(x_i))$, either the specification or the program should be modified until $B(RD_f(x_i)) = B(RD_p(x_i))$.

3) Develop test case according to $B(RD_f(x_i))$ or $B(RD_p(x_i))$.

Each $B(RD_f(x_i))$ or $B(RD_p(x_i)) \quad (i = 1,2,\cdots,n)$ has *2* boundary points. Test cases take three values for each boundary point, that is, the boundary point itself, boundary point + ε, and boundary point - ε, where ε is a small amount. So, *2*3* test cases are produced for each input variable x_i. As a result, the testing approach only needs *2*3*n* test cases, where *n* is the number of input variables. However, in boundary-interior path testing, 6^L test cases are required in the worst case, where *L* is the number of loops of the program, on the assumption that each loop contains an *IF-THEN-ELSE* statement. In domain testing, *2*N*M* test points are required in the worst case, where *N* is the number of program paths, and *M* is the number of predicates on a chosen path. Moreover, they provide no guideline for selecting actual values with which to execute a chosen path. It is obvious that our domain analysis testing approach enjoys the advantage of fewer test cases than path testing and domain testing, and the test generation is independent on path selection.

4. Program operational domain D_p

The determination of program operational domain is a foundation for the domain analysis testing approach presented in this paper. Hence, we develop an automated determination system of operational domain, called *ADSOD*, whose input is a program written in C programming language, and the output is its operational domain D_p.

First, we introduce some basic concepts and then show their usage.

Each program statement is regarded as a node in our discussion. Let *D(v)* be a set of nodes in which variable *v* is defined, and *U(v)* denote a set of nodes in which variable *v* is used. Considering those variables that are related to the definition and use of variable *v*, we design two sets, *inD(v)* and *inU(v)*. The set *inD(v)* consists of all variables referenced when variable *v* is defined, and *inU(v)* is composed of all variables referenced when variable *v* is used. For example, suppose there are following assignment statement 1) and condition statement 2) in a program:

1). $v = x + y - z$ 2). if $(v > x+y)$ …

Then, *inD(v)* corresponding to statement 1) is {*x,y,z*}, and *inU(v)* corresponding to statement 2) is {*x,y*}.

Definition 3: *Node correlating with variable*

Let k be a node, and v be a variable. By node k correlates with variable v, represented by R(k,v), we imply that k and v satisfy at least one of the following conditions.

(1) $k \in D(v) \vee k \in U(v)$.

(2) $k \in D(x)$, $x \in (\, inD(v) \vee inU(v)\,)$.

(3) $k \in D(x)$, $\exists y_1, y_2, \cdots, y_s$ such that

$y_1 \in (inD(v) \vee inU(v))$, $y_2 \in inD(y_1), \cdots, y_s \in inD(y_{s-1})$, and $x \in inD(y_s)$, where $x, y_1, y_2, \cdots, y_s$ are program variables.

Condition (1) shows that node *k* is directly related to variable *v,* namely variable *v* is defined or used in node *k.* Condition (2) indicates that node *k* connects with variable *v* via variable *x*, i.e. variable *x* is defined in node *k* and is referenced when defining or using variable *v*. Condition (3) denotes that there exist variables $x, y_1, y_2, \cdots, y_s$ such that node *k* is indirectly related to variable *v*. In the cases (2) and (3), we also say that variable *x* correlates with variable *v*.

A *specific expression* ***e*** is the one that the variables in ***e*** are related to input variable and ***e*** can not take some illegal values due to some constraints, such as when ***e*** is used as a divisor or as a parameter of some standard subroutines, e.g., $sqrt(), \log(), a\cos(), a\sin()$, etc. For example, the expression $f(x1+1.0)-f(3.453)$, in statement 10 of program *P* in Fig.4, is a specific expression. Its value cannot be equal to zero to avoid divide-by-zero failure.

```
1    float f(float x)
2    {
3        float   y;
4        y=((x-0.5)*x+16.0)*x-80.0;
5        return(y);
6    }
7    float xpoint(float x1)
8    {
9        float y;
10       y=x1*f(x1)/(f(x1+1.0)-f(3.453));
11       return(y);
12   }
13   void main()
14   {
15       float x,y;
16       scanf("%f",&x);
17       y=xpoint(x);
18       printf("%f",y);
19   }
```

Fig. 4   Program *P*

Definition 4: *Expression equivalence*

We define a mapping **exec***, which maps a program P, a specific expression* **e***, and program input x to a value of the expression, namely*

$$Pset \times Eset \times Xdomain \rightarrow Erange$$

$$i.e. \qquad exec(P,e,x) = e\_value\ ,$$

where Pset is a set of programs, Eset consists of specific expressions in a given program, Xdomain is made up of all values that input x can hold, and Erange contains all values that a chosen expression can take.

Program P and P′ $(P, P' \in Pset)$ are equivalent to each other with respect to expression ***e*** *(**e**∈ Eset) only if on all input x (x∈ Xdomain of P and P′) the following formula holds*

$$exec(P,e,x) = exec(P',e,x)\ .$$

4.1 Automated determination system of operational domain *ADSOD*

To determine the operational domain of a program, a lot of related information, for instance, definition, use and type description of a variable, the correlation among variables, etc, need to be taken into account. For this purpose, we firstly build various tables for each procedure of the program, including parameter table, variable table, input variable table, specific expression table and so on. Parameter table records information connected with a procedure, such as the procedure name, parameter names, parameter types, the numbers of variables and input variables, and called procedures, etc. Variable table is the most important data structure in the system *ADSOD*. It stores the names and types of variables defined in the procedure and their associated sets, i.e. $D(), U(), inD()$ and $inU()$. Input variable table writes down the names of input variables appearing in the procedure. Meantime, detailed information about definitions and uses of input variables is contained in the corresponding variable table. Specific expression table saves the nodes corresponding to the expressions and the variables involved in the expressions.

Obviously, each predefined type has a domain assigned to it. For example, an input variable of *unsigned short* type, in C programming language, can take values from 0 to 65535. The domain *Dx* of input variable *x* of predefined type, such as *int*, *short*, *unsigned long,* etc, can be computed by analyzing its definitions, uses, type description and the correlation among variables with the help of above various tables. For an input variable *x* of structure type, its domain *Dx* may be calculated based on the domains of individual fields. For an input variable *x* of pointer type, its *Dx* may be obtained according to the type of the object it points to. For an input variable *x* of character string type, assume that *x* is related to a character string variable *v* and some string subroutines such as $atoi(), atof(), strcpy(), strcat()$, etc, for example, suppose that there is the following code fragment in a program:

strncpy(v, x,5); /* copy initial 5 characters from *x* to *v* */
i = atoi (x); /* convert *x* to integer */

where *i* is a variable of *int* type, then, the domain *Dx* can be determined by variable *v* and *i* and the subroutines $strcpy()$ and $atoi()$.

But, if an input variable *x* is involved in a specific expression ***e,*** for instance, the expression

$f(x1+1.0)-f(3.453)$ in Fig.4, which is used as a divisor and *f(x)=((x-0.5)*x+16.0)*x-80.0*, it is difficult to calculate its domain *Dx* by using static analysis. Therefore, we construct a facilitation procedure, denoted by *based_on_expression_e$_r$(),* to identify the illegal values of input variable with respect to the expression*,* where *r* is the serial number of the procedure corresponding to r^{th} specific expression.. As a result, all values that input variable *x* can take, i.e. *Dx,* can thus be determined.

Fig. 5 displays the structure of *ADSOD* system. It consists of two modules *pro-processor* and *domain_determination().* First of all*,* the *pro-processor* is derived to create the various tables and to construct the procedures *based_on_expression_e$_r$()* for specific expressions, if there are specific expressions in the program. Secondly, the procedures are complied together with original program *P* to generate executable codes. Then, the domain Dx_i $(i=1,2,\cdots,n)$ is computed by invoking the module *domain_determination().* A cross product of all domains Dx_i constitutes the program operational domain D_p, namely $D_p = Dx_1 \times Dx_2 \times \cdots \times Dx_n$.

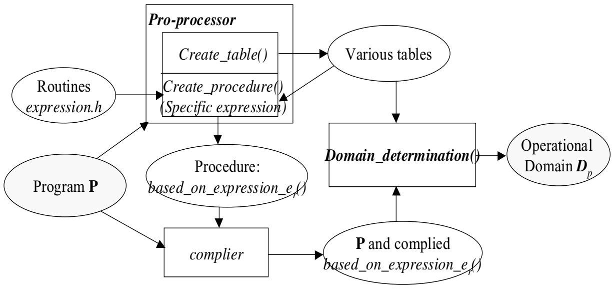

Fig.5 Structure of ADOSD

4.2 The construction of the procedure *based_on_expression_e$_r$()*

In order to identify illegal values of input variables in a specific expression ***e***, we construct a procedure, *based_on_expression_e$_r$(),* with respect to the expression ***e*** by using program slicing technique. The procedure is made up of a slice and some instrumentation statements.

A slice is defined by a slicing criterion, $C=(I,V)$, where *I* is a node of program *P* and *V* is a subset of variables in *P*. Given criterion *C*, a slice of program *P* is composed of all nodes in *P* whose execution may affect the value of variables in set *V* at *I* [12,13]. To produce a slice for a specific expression ***e***, a slicing criterion $C=(I,V)$ needs to be designed, in which *I* is the node containing the expression ***e***, and *V* is a set of variables in ***e***. According to the criterion *C,* a program part P' can be generated by program slicing. It contains all statements preceding node *I* that directly or indirectly influence variable $v, v \in V$, at *I*. Meanwhile, *P'* and *P* should be equivalent to each other with respect to the expression ***e*** on all input *x* of *P* and *P'*.

A specific expression ***e*** may be used as a divisor or as a parameter of some standard subroutine such as $sqrt(), \log(), a\cos(), a\sin()$, etc. We develop some routines for each instance in advance, and set them into corresponding header file, e.g. *expression.h* in Fig.5. For example, the routine *fdivi()* is designed to calculates the values that make a general float function equal to zero, and the routine *isqrt()* is developed to compute the values that make a general integer function smaller than zero. We connect the general functions with concrete functions via statement $\#define$, in C programming language, and insert some instrumentation statements into program part *P'* to form a integrated and executable procedure *based_on_expression_e$_r$().* The algorithm is shown in Fig.6. Fig.7 gives a procedure *based_on_expression_e$_1$()* with respect to a specific expression e: $f(x1+1.0)-f(3.453)$ in Fig.4, where corresponding instrumentation statements are shown in italics, and *float_func()* stands for a general float function. The illegal values of input variables *x* for the expression ***e*** can be identified by executing the procedure. That is, 2.453.

```
create_procedure( input: e, I ; output: name)
{
  // The input e is a specific expression of P //
  // I is the node corresponding to e. //
  // The output is the name of the procedure
     based_on_expression_e_r( ).//
 1. Collect all variables in e into set V,
    $V=\{v_1,v_2,\cdots,v_m\}$ .
 2. Let $N=\phi$ , $j=1$ , $v_j \in V$
    // N is a set of nodes that are correlated with
       variable $v_j$. //
 3. Find out nodes k satisfying $R(k,v_j)$ , and put
    into the set K, $K=\{k_1,k_2,\cdots,k_z\}$ .
 4. While ( $k_t \in K \wedge k_t \le I, 1 \le t \le z$ ) $N=N \cup \{k_t\}$
 5. $V \leftarrow V-\{v_j\}, j=j+1$
 6. If $(V \ne \phi)$ go to 3.
 7. Extract out corresponding statement from P to
    form program part $P'$ according to nodes in N.
 8. Insert instrumentation statements into $P'$ to form
    based_on_expression_e().
}
```

Fig.6. The algorithm of constructing the procedure *based_on_expression_e$_r$( )*

```
float f(float x)
{
    float y;
    y=((x-0.5)*x+16.0)*x-80.0;
    return(y);
}
float xpoint(float x1)
{
    float y;
    y=f(x1+1.0)-f(3.453);
    return(y);
}
#define float_func xpoint
#include expression.h
void based_on_expression_e1( )
{
    fdivi( );
}
```

Fig.7 *based_on_expression_*e_1*( )*

To verify the validity of the *ADSOD* system, a number of different kinds of practical programs have been executed. Although each program is made up of only dozens of statements, they contain lots of common structures, such as complicated control relations, combinations of various input variables, specific expressions, etc. For example, ***MaxMin*** program, which will be detailedly discussed in Section 5, has two input variables. One is an integer variable *argc*, the other is character string variable *argv*. The operational domain of ***MaxMin*** program can be determined when the *ADSOD* system is invoked.

Similarly if we apply the *ADSOD* system to program *P* shown in Fig.4, *ADSOD* also correctly computes its operational domain, that is, the input variable *x* must not be assigned 2.453. Experimental results illustrate that *ADSOD* supports not only the determination of input domain of integer and real data types, but also non-numeric data types such as characters string types. The accomplishment of *ADOSD* provides us a facility for test data generation based on the input domain analysis of specifications and programs.

5. Example study

In this section, we compare the effectiveness of our domain analysis testing approach with partition testing strategies in fault detection with an example ***MaxMin,*** which is a variation of the program taken from reference [14].

5.1 Specification and program of *MaxMin*

MaxMin prints the maximum and minimum of keyboard-input integer arguments. There is an option "*–ceiling*". Two ceilings are provided immediately after *"-ceiling*", denoted by *CEIMIN* and *CEIMAX,* respectively. If the minimum is smaller than *CEIMIN*, *CEIMIN* becomes the resulting minimum. If the maximum is larger than *CEIMAX*, *CEIMAX* becomes the resulting maximum. If the argument after ***MaxMin*** begins with a '-' but not *"-ceiling*", ***MaxMin*** prints an error message. Fig.8 lists the major fragment of ***MaxMin*** program, and Fig.9 displays its control flow graph.

```
1 #define BUFSIZE 20
2 void main(int argc,char **argv)
4 { char CEIMAX[BUFSIZE];
5   char CEIMIN[BUFSIZE];
6   char tempstr[BUFSIZE];
7   long lmax, lmin;
8   long resultmax, resultmin, tempvar;
9   initialize();
12   for(argc--,argv++;argc>0&&'-'==**argv;argc--,argv++)
14   { if (!strcmp(argv[0],"-ceiling"))
16      {   strncpy(CEIMIN, argv[1],BUFSIZE);
17          lmin=atol(CEIMIN);
18          argv++;
19          argc--;
20          strncpy(CEIMAX, argv[1],BUFSIZE);
21          lmax=atol(CEIMAX);
22          argv++;
23          argc--;}
25      else
26      {
27         printf("Illegal option %s.\n",argv[0]);
28         exit(2); } }
31   if (argc==0)
32   {
33      printf("Requires at least one argument.\n");
34      exit(2); }
36   for (;argc>0;argc--,argv++)
37   {
38       strcpy(tempstr,argv[0]);
39       tempvar=atol(tempstr);
40       if(tempvar>resultmax)
41               resultmax=tempvar;
42       if(tempvar<resultmin)
43               resultmin=tempvar; }
45   if(lmax<resultmax)
46         resultmax=lmax;
47   if(lmin>resultmin)
48         resultmin=lmin;
49   printf("resultmin %ld\n",resultmin);
50   printf("resultmax %ld\n",resultmax);
51   exit(0);}…
```

Fig.8 ***MaxMin*** program

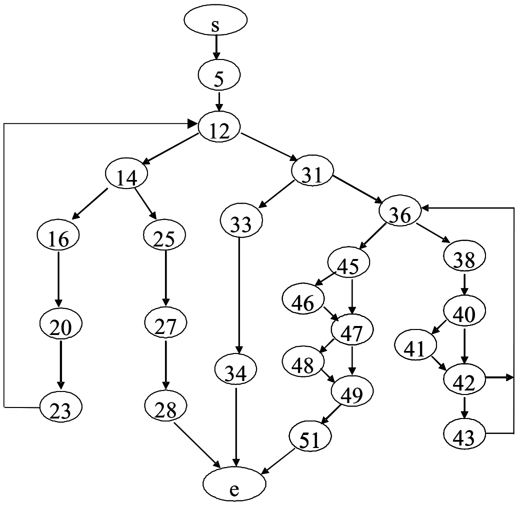

Fig. 9 Control Flow Graph of ***MaxMin***

5.2 Test generation based on the partition testing strategies

There are 258 boundary-interior paths according to the control flow graph of ***MaxMin*** program, but only 30 paths are feasible. We generate 30 test cases as to these feasible paths, denoted by *test_set_1*$=\{T1, T2, \cdots, T30\}$, execute ***MaxMin*** program with these test cases, and find that the responses of test cases T7, T8, T25, T26, T27, T28, T29, T30 are different from the expected ones.

Compared with the correct responses, T7 and T26 go wrong since the following argument after *CEIMAX* is a negative integer beginning with '-'. T8 and T27 are serious control flow faults. The code takes place *Core dumped* when *CEIMIN* or *CEIMAX* are not given. T25, T28, T29 and T30 are incorrect due to defining "*–ceiling*" two times.

As a result, these test cases reveal three software faults in ***MaxMin***:

Fault 1). Line14: Checking option "-ceiling" should be put down before the line 12.

Fault 2). Line16: The exit of error should be added when arg[1] is not given.

Fault 3). Line20: The exit of error should be added when arg[1] is not given.

Then, we apply the simplified domain testing strategy to develop *ON-OFF* test points for each border of each boundary-interior path domain. The test cases of T1 to T30 can be used as *ON* test points corresponding to each path. *OFF* points can be fixed by selecting as close as possible to the *ON* points but not satisfying the path condition associated with each border. Although some *ON* or *OFF* points may be used byproduct to check other border, in the worse case, there are 30+312=342 *ON-OFF* test points. However, they only detect the same three software faults.

5.3 Test generation based on the domain analysis testing approach

The operational domain D_p of ***MaxMin*** program can be obtained by calling the *ADSOD* system. The results are as follows:

The value of input variable argc *is ≥1.*

The maximal number of characters that input variable argv *allows to enter is BUFSIZE-1.*

The values that variable argv *can hold are between -9223372036854775808 and 9223372036854775807.*

This indicates that input variable *argc* cannot be smaller than 1, input variable *argv* should satisfy $0 \le \# \arg v \le BUFSIZE - 1$, and the values of *argv* should be between -9223372036854775808 and 9223372036854775807. The buffer size, i.e. *BUFSIZE,* defined in ***MaxMin*** is 20, and the last character of the buffer must be a terminating symbol. Thus, the variable *argv* is at most with 19 characters, i.e. $0 \le \# \arg v \le 19$.

By the definition of the set of boundary test case of domain D_p, we have $T(D_p) = B(RD_p(\arg c)) \cup B(RD_p(\arg v))$. The center point of the projection of domain D_p on *argv* axis is 10, i.e. $a_{\arg v} = 10$. The upper bound of variable *argc*, denoted by *M,* is not given, but it can be automatically calculated by the number of characters of keyboard-input and that of each *argv*. Here, we suppose that $a_{\arg c} = 10$. Then, $T(D_p) = \{\{1, M\}, a_{\arg v} = 10\} \cup \{a_{\arg c} = 10, \{0,19\}\}$.

The specification of ***MaxMin*** program specifies neither the limitation to variable *argc,* nor the limitation to the number of characters that variable *argv* can enter and the values that *argv* can hold. Thus, its $T(D_f)$ is uncertain. So, $T(D_p) \ne T(D_f)$. Consequently, the specification should be described in more detailed. We add following pre-conditions into the specification.

Pre-condition1: The value of input variable argc *is ≥1.*

Pre-condition2: *The maximal number of characters that input variable* argv *allows to enter is BUFSIZE-1.*

Pre-condition3: The values that variable argv *can hold are between -9223372036854775808 and 9223372036854775807.*

Thus, $T(D_p) = T(D_f)$.

Following test requirements can be produced according to $T(D_p)$.

1. Let *argc*=10, check the boundary of input variable *argv*, namely its number of characters and maximal and minimal value.

R1: *MaxMin* with argument of 0 character.

R2: *MaxMin* with argument of 1 character.

R3: *MaxMin* with argument of 19 characters.

R4: *MaxMin* with argument of 20 characters.

R5: *MaxMin* with value –9223372036854775808

R6: *MaxMin* with value < –9223372036854775808

R7: *MaxMin* with value 9223372036854775807

R8: *MaxMin* with value > 9223372036854775807

R9: *MaxMin* with *CEIMIN* / *CEIMAX* of 0 character.

R10: *MaxMin* with *CEIMIN*/ *CEIMAX* of 1 character.

R11:*MaxMin* with *CEIMIN*/*CEIMAX* of 19 characters.

R12: *MaxMin* with *CEIMIN*/*CEIMAX* of 20 characters.

R13: *MaxMin* with *CEIMIN* / *CEIMAX* of value –9223372036854775808

R14: *MaxMin* with *CEIMIN* or *CEIMAX* of value 9223372036854775807

R15: *MaxMin* with *CEIMIN* / *CEIMAX* of value < –9223372036854775808

R16: *MaxMin* with *CEIMIN* or *CEIMAX* of value > 9223372036854775807

Among them, R9-R16 inspects the boundary of *CEIMIN* or *CEIMAX* when the option "*-ceiling"* is given.

2. Let *#argv*=10, check the boundary of input variable *argc*, namely the number of arguments.

R17: *MaxMin* with 0 arguments.

R18: *MaxMin* with 1 argument.

R19: *MaxMin* with as many arguments as possible.

On the basis of these test requirements, we develop 15 test cases, represented by $test\_set\_2 = \{T1', T2', \cdots, T15'\}$, in which some test cases satisfy more than one test requirements. Then, running ***MaxMin*** program with these test cases, we find that the responses of test cases T2′, T4′, T5′, T8′, T9′, T11′ and T12′ are different from the expected ones. Compared with the correct responses, T2′ and T8′ are not correct due to entering 20 characters, while the specification requires that only 19 characters be taken into account. T4′ and T11′ go wrong since there exist the argument whose value is smaller than –9223372036854775808, so do T5′ and T12′ because of larger than 9223372036854775807. T9′ is incorrect since the argument following *CEIMAX* begins with character '-'. More detailed discussion is described in [15].

As a result, the test cases expose five software faults. They are:

Fault 4). Line 16: The BUFSIZE should be BUFSIZE –1.

Fault 5). Line 20: The BUFSIZE should be BUFSIZE –1.

Fault 6). Line 38: The BUFSIZE should be given.

*Fault 7). The value of arguments should be between –*9223372036854775808 and 9223372036854775807.

Fault 8). Line14: Checking option '-ceiling' should be put down before the line 12.

In these faults, only *fault 8)* is same as the *fault 1),* which is detected by the partition testing strategy too.

5.4 Result analysis

Code coverage has been known to be an important metric for testing software [16,17]. We measure the coverage of *test_set_1*, *test_set_2* and *test_set_1* + *test_set_2* using ATAC (***A****utomatic* ***T****est* ***A****nalysis for* ***C***) tool [18]. The results are show in Fig.10. The coverage of *test_set_1* is same as that of *test_set_1*+ *test_set_2,* but it only finds three faults in ***MaxMin*** program. This demonstrates that two test sets with identical coverage could have very different fault detection effectiveness. Moreover, the coverage of *test_set_2* is smaller than that of *test_set_1,* but *test_set_2* finds five faults. Thus, the coverage itself of a test set is not a reliable indication for fault detection effectiveness. Even if the highest coverage is achieved, structural testing may still omit some software faults.

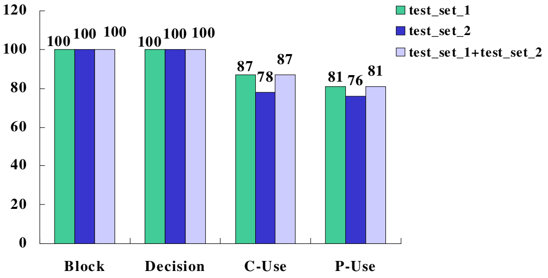

Fig.10 Coverage compare of two test sets

6. Conclusions

In this paper, we present an innovative software testing approach based on input domain analysis of specifications and programs, and show how to generate test cases according to the set of boundary test case of functional domain and operational domain. Preliminary experimental results indicate that the new testing approach is highly effective in detecting the faults related to small shifts in input domain boundary, and is more economical in test case generation than the partition testing strategies.

This domain analysis testing approach is different from boundary value analysis, which belongs to partition testing strategies. Boundary value analysis, according to an identified equivalence class, produces test cases that lie close to a sub-domain boundary to check the program behavior. The domain analysis testing approach, on the other hand, generates test cases by verifying the coincidence of operational domain and functional domain to find the faults that are resulted from the discrepancies between these two domains.

As to path testing and domain testing strategies, their effectiveness of fault detection is limited because they lack a path selection criterion to guide choosing a faulty path to test. Moreover, structural testing may still omit some software faults even if the highest coverage is achieved. The domain analysis testing approach, however, does not suffer the problem since both requirement specifications and implementation details are taken into account at the same time when testing a program. Besides, the domain analysis testing requires fewer test cases than the structural testing. As we have seen from the experiment, it is difficult to detect all software faults with a single testing technique. The strengths and weaknesses of software testing techniques are somewhat complementary. Consequently, it is possible to combine our domain analysis testing approach with the partition testing strategies to achieve higher software testing effectiveness.

Acknowledgement

The work described in this paper was fully supported by the Hong Kong Research Grants Council, under Project No. CUHK 4360/02E and Young Science Foundation of BUCT, China, under Project No. QN0312.

References

[1] B. Beizer. "Software Testing Techniques," *International Thomson Publishing Inc.*, 2nd edition, 1990.

[2] P. Maxwell, I. Hartanto and I. Bentz. "Comparing Functional and Structural Tests". *Proceedings of International Test Conference, 2000*, pp. 400-407.

[3] S. C. Ntafos. "A comparison of Some Structural Testing Strategies," *IEEE Transactions on Software Engineering,* Vol. 14, No. 6, June 1988, pp.868-874.

[4] I. Keidar, R. Khazan, N. A. Lynch, A. A. Shvartsman. "On fault classes and error detection capability of specification-based testing," *ACM Transactions on Software Engineering and Methodology*, Vol. 11, No.1, January 2002, pp. 58-62.

[5] A. J. Offutt and S. Liu. "Generating Test Data from SOFL Specification," *The Journal of System and Software,* 49(1), December 1999, pp. 49-62.

[6] S. C. Reid, "An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing", *Proceedings of Fourth International on Software Metrics Symposium,* 1997, pp. 64-73.

[7] P. C. Jorgensen. "Software Testing: A Craftsman's Approach". *CRC Press LLC*. 2002.

[8] B. Jeng and E. J. Weyuker. "A Simplified Domain-Testing Strategy," *ACM Transactions on Software Engineering and Methodology*, Vol.3, No.3, July 1994, pp254-270.

[9] M. R. Lyu, Editor, "Handbook of Software Reliability Engineering", *IEEE Computer Society Press*, CA, 1996, p. 850.

[10] W. J. Gutjahr, "Partition Testing vs. Random Testing: The Influence of Uncertainty," *IEEE Transactions on Software Engineering*, Vol.25, No.5, Sept. /Oct. 1999, pp.661-674.

[11] B. Korel. "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, Vol.16, No.8, August 1990, pp. 879-879.

[12] F. Tip. "A Survey of Program Slicing Techniques", *Journal of Programming Languages*, Sept.1995, 3(3), pp. 121-189.

[13] T. Gyimothy, A. Beszedes and I. Forgacs. "An Efficient Relevant Slicing Method for Debugging," *Software Engineering Notes*, vol.24, No.6, 1999, pp.304-312.

[14] B. Marick. "The craft of software testing," *PTR Prentice Hall*, NJ, 1995.

[15] R. Zhao, "Research on Software Testing Methodologies", *Ph.D. thesis*, Chinese Academy of Science, 2001.

[16] T. W. Williams, M. R. Mercer, J. P. Mucha and R. Kapur. "Code Coverage, What Does it Mean in Terms of Quality?" *Proceedings of Annual on Reliability and Maintainability Symposium,* 2001, pp. 420-424.

[17] M. Chen, M. R. Lyu, and E. Wong, "Effect of Code Coverage on Software Reliability Measurement", *IEEE Transactions on Reliability*, Vol. 50, No. 2, June 2001, pp.165-170.

[18] M. R. Lyu, J. R. Horgan, and S. London, "A Coverage Analysis Tool for the Effectiveness of Software Testing", *IEEE Transactions on Reliability*, Vol. 43, No. 4, December 1994, pp. 527-535.