An Approach for Domain Reduction with Data Dependence in Mutation Testing

Gaochao Xu
Department of Computer Science and Technology, JiLin University, Changchun, China
Email: xugc@jlu.edu.cn

Yushuang Dong, Xiaodong Fu, Yan Ding, Jia Zhao and Xinzhong Liu
Department of Computer Science and Technology, JiLin University, Changchun, China
Email: yushuangdong@gmail.com, fuxd@jlu.edu.cn, dingyan11@mails.jlu.edu.cn, zhaiyj049@sina.com, lxz_jlu@163.com

***Abstract*—As a testing strategy to evaluate the completeness of test cases, mutation testing has been identified as a "fault-oriented" technique for unit testing, which is mainly used to generate complete test cases. The path-oriented technique of test data generation is a highly efficient technique which implements test data generation by building and solving constraint systems. Most of path-oriented generation techniques only take control dependence among statements into consideration, which is to build constraint system by analyzing control flow graph. However, it neglects the influence of data dependence among statements on constraint system. Therefore, this paper improved test data generation technique of domain reduction and proposed a new domain reduction method with data dependence. It added detecting of equivalent mutants and solved influences on constraint systems caused by multiple conditional branch statement. Experimental results showed that this method improved success rate and execution efficiency of test data generation in a significant extent.**

***Index Terms*—Mutation testing, Equivalent mutant, Constraint system, Automatically software testing, Test data generation**

I. INTRODUCTION

Mutation testing is a kind of "fault-oriented" technique for unit testing [1,2]. Mutation testing is based on two basic hypotheses [1,3]: (1) Coupling effect hypothesis. (2) Competent programmer assumption. Based on this assumption, mutation testing uses mutation operator [4-7] to syntactically change a simple test program statement, and automatically generates sets of error procedures corresponding to original program which were known as mutants. When we Use existing test cases in mutation testing, if mutants' output is different from original program's output or mutants stop running before the end of program, the mutants can be killed by test cases, then reselect the other live mutants to do the same operation until all mutants are killed or attain a certain mutation score [4].

For a program P and a set of test data T, the number of mutants is M, the number of dead mutants is D, and the number of equivalent mutants is E. The mutation score is defined to be:

$$MS(P, T) = D / (M - E).$$

P is a program, M is a mutant of P on statement S, and t is a test case for P. If M can be killed by the test case t, we call t is valid, otherwise t is invalid. To kill M, t needs three broad characteristics [15,25]: (1) Reachability; (2) Necessity; (3)Sufficiency. Sufficiency will not be considered in this paper.

There are three main reasons for the mutants not to be killed: (1) The set of test cases is not complete enough, and need to generate new test data to kill the mutants; (2) The mutants are functionally equivalent to the original program, viz, equivalent mutant. The equivalent mutant accounted for about 10% of the total number of mutants [8,9]; (3) Data values of mutant statement is shielded or application of software error correction technology lead to the fact that the mutants can't be killed, and it is an undecidable problem [10].

We have given a test data generation method in reference [11]. But this method did not consider the influence caused by equivalent mutant. If the mutant is equivalent mutant, the mutant cannot be killed. So we give a method to detect equivalent mutant in test data generation process. In addition, we solved the problem of the multiple conditional branch statement.

II. RELATED WORKS

TDG (Test Data Generation) is defined as selecting appropriate test standard and input process for identification which satisfies requirements [8]. TDG has been widely applied by industrial circles such as web applications testing [12], flight simulator fidelity evaluation [13], and etc. According to generate way, TDG can be divided into random, path-oriented, and goal-oriented test data generation [14]. Random TDG is a simple method, but random approach has low probability in finding semantically small faults. Path-oriented TDG includes constraint-based TDG (CBT) [4], dynamic domain reduction TDG (DDR) [15], and an improved method presented in reference [16]. Path-oriented

Corresponding author: Jia Zhao.

doi:10.4304/jsw.8.6.1511-1517

approach has better prediction of coverage, however, it is more difficult to find test data and it often selects infeasible paths. In these three path-oriented approaches, CBT is a highly effective TDG method. However, CBT can't handle loop statements depending on input variables and array indexes. It is likely to cause a solvable constraint system to become unsolvable [15,16,18]. DDR introduces dynamic translations of control flow graph (CFG) to solve the problem of loop, arrays and pointers that CBT cannot handle. DDR is more effective than CBT in time and space aspect [15,16]. Reference [16] combines the CBT and DDR. It reduces the amount of TDG by assembling not contradictory necessity conditions in constraint system. It solves constraint system through iterative relaxation method [19]. Goal-oriented TDG includes chaining approach for TDG (CAT) [8], and assertion-oriented approach [8,17]. Goal-oriented approach is difficult to predict the range of the coverage but more flexible to find test data and reduce the probability of selection relatively infeasible paths. CAT considers describes both control flow and data flow as event sequence, it improved the testing data discovery rate. There are also many other optimize on TDG. Reference [20] proposes a micro-kernel engine to optimize the automatic test system and reference [21] proposes a software reliability test method based on Markov usage model. Many methods use the genetic algorithm to optimize the TDG [22,23,24]. Reference [25] proposed a reusing test cases method because half of the code of the software systems written today is to produce the required GUIs.

III. Constraint-based Mutation Testing

Figure 1 is a sample program and the corresponding control flow graph. For example of Figure 1, a set of nodes is $N = \{1, P1, P2, P2, 3, 4, 5, P3, P4, P5, P6, P7, 14, 15\}$, a set of edge in N is $E = \{(1, P1), (P1, P2), (P2, 3), (3, 4), (4, 5), (5, P3), (P3, P4), (P4, P5), (P5, P6), (P6, P7), (P7, 14), (14, 15)\}$, unique entry is node 1, and unique exit is node 15, CFG is $G = (\{1, P1, P2, P2, 3, 4, 5, P3, P4, P5, P6, P7, 14, 15\}, \{(1, P1), (P1, P2), (P2, 3), (3, 4), (4, 5), (5, P3), (P3, P4), (P4, P5), (P5, P6), (P6, P7), (P7, 14), (14, 15)\}, 1, 15)$. A set of Input variables is $I = (a, b, c)$. Input domain is $D = D_a \times D_b \times D_c$. TDG is to find $a \in D$, $b \in D$, $c \in D$ to execute the program. Branch prediction describes the transfer condition for each edge in G such as $BP(P2, 3)$ = "$a > b$". No transfer condition branch predicate expression is empty such as $BP(4, 5)$ = "". A path in CFG from node $P1$ to node 14 is a sequence $P = <P1, P2, P2, 3, 4, 5, P3, P4, P5, P6, P7, 14>$. We mark P^* if P in loop statements. If there is a program input I traversed P, we called P as feasible path, and otherwise P is infeasible path. Usage set of variables in node 4 as $U(4)=\{a, b\}$. Definition set of variable in node 4 is $D(4) = \{a\}$, path $P = <N_i,\ldots, N_j>$ from node N_i to node N_j is a definition-clear path to partial set of variables S if S is not modified except for node N_i and node N_j, $\forall N_{ki} \in P \bullet 1 < i < q \wedge v \notin D(N_{ki})$. If there is a definition-clear path $P = <N_i,\ldots, N_j>$ to S and $D(N_i) \cap U(N_j) \neq \varnothing$, we called node N_i and node N_j is DU-Pair, node N_i is dependent on node N_j.

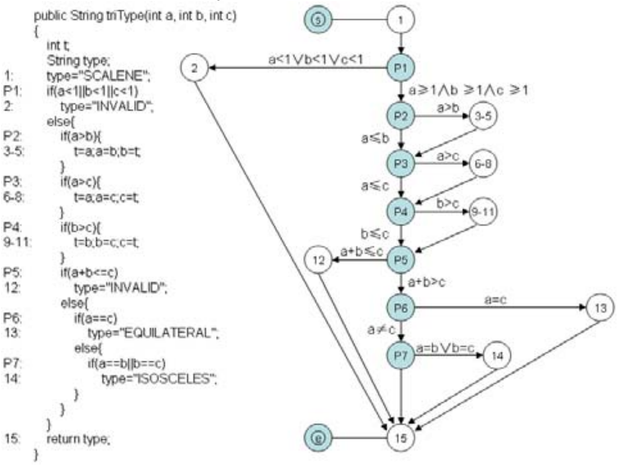

Figure 1. A sample program and the corresponding control flow graph.

Constraint-based TDG technology, including CBT, DDR and the method introduced in reference [16], has two key problems: (1) Constraint system building; i.e., building the constraint system by reachability condition and necessity condition; (2) Constraint system solving, which tells how to automatically generate test data. These three methods have better efficiency in execution, but they still have a shortage that the whole generation process is only guided by CFG. It has two influences because of its own information limitations: (1) Path-oriented TDG must generate a great deal of paths, and judge and choose them until a valid path is obtained, and it is fatal to complex system; (2) It could cause no solutions because of the difficulty of achieving the destination node if ignoring data dependence.

For improving goal-oriented and path-oriented methods, we combine the goal-oriented CAT method and present an approach for domain reduction with data dependence (DRD). It added the detecting of equivalent mutants and solved the problem of the multiple conditional branch statement. It uses the path with data dependence (PDD). The last variable definition set in node n is $LD(P5)=\{4, 5, 7, 8\ 10, 11\}$ in figure 1. The set of data dependence with node $P5$ for path $P = <1, P1, P2, 3, 4, 5, P3, P4, P5, 12>$, we get $LDP(P5) = \{3, 4\}$, $LDP(5) = \{3\}$. PDD is an extended representation of the path. It not only contains the control information of the path, but also data dependence information of all nodes. We marked the node n and the last variables definition set with $n{:}LDP(n)$, PDD is a sequence of node $n{:}LDP(n)$. For example the path we mentioned above can be expressed as $<1, P1, P2, 3, 4, 5{:}\{3\}, P3{:}\{4\}, P4{:}\{5\}, P5{:}\{4, 5\}, 12>$. We call PDD is a complete PDD if all adjacent nodes $<N_i, N_j>$ in PDD satisfy $<N_i, N_j> \in PDD \wedge (N_i, N_j) \in E$, otherwise the PDD is a part PDD.

IV Domain Reduction Considering Data Dependence

DRD begins with the analysis of the program. It generates CFG, and initializes the state of PDD as $<N_s$,

N_g>. First, use the mutation operator on target node N_g to generate a set of mutation. Then combine the necessity condition C_n collection marked by C, and build the constraint system for each element of the necessity condition collection C, and automatically detect equivalent mutants, and solve the constraint system to generate the test data at last. PDD generation is the key to build constraint system. The data process flow graph of DRD is shown in figure 2, and its sequence operator may refer to Z language [27]. Process can be divided into four phases: (1) pretreatment; (2) constrain system building; (3) detecting equivalent mutants; (4) constraint system solving.

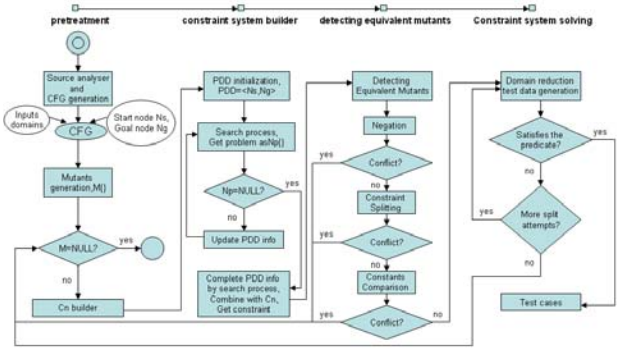

Figure 2. Dataflow diagram

DRD improved the process of constraint system building. Constraint system building can be described as follows: dynamic search the PDD from the start node N_s to the goal node N_g, get the path P with considering data dependence, build the reachability condition C_r by P, combine with the necessity condition C_n and build the constraint system at last. There are three key points: (1) two phases of PDD generation process; (2) search process related to generate process; (3) build the constraint system by PDD. It is described as follows.

A. PDD Generation

Set PDD_0 = <N_s, N_g>. PDD generation includes key PDD generation and complete PDD building. Search process calculates $LD(p)$, set $LD(p) = \{N_{d1}, N_{d2}, \ldots, N_{dn}\}$, generates n PDDs. Each new PDD generated with data dependence to the original PDD. N_{dn} is empty node, we marked as $N_{dn}=\varepsilon$. We just add the problem node to new PDD. The pattern of new PDD is:

$$PDD_i = \begin{cases} < N_s, p, N_g > & , \quad i = n \\ < N_s, N_{di}, p:\{N_{di}\}, N_g >, & 1 \le i < n \end{cases}$$

Search process selects a PDD_i from the new PDDs, and generates the next problem node p_i and calculats $LD(p_i)$, if $N_{fm} = \varepsilon$, set $LD(p_i)=\{N_{f1}, N_{f2}, \ldots, N_{fm}\}$ and generates m PDDs by using $LD(p_i)$. The pattern of new PDD is:

$$PDD_{ik} = \begin{cases} < N_s, p_i, N_{di}, p:\{N_{di}\}, N_g > & , \quad k = m \\ < N_s, N_{fk}, p_i:\{N_{fk}\}, N_{di}, p:\{N_{di}\}, N_g >, & 1 \le k < m \end{cases}$$

There is $i = n$ similar to this. If search process found no new problem node, stop the key PDD generation. All generated PDD can be viewed as a tree and new PDD as the child node generated from PDD_0 = <N_s, N_g> through the problem node and element of last definition, and generate the problem node of the child node by search process. Continue the process until all child nodes of the PDD have been found. The searching process is shown as figure 3. From another point of view, PDD generation can be seen as depth-first searching process for searching tree, and it is a sequence of program which can reach goal node.

New node cannot be included in that has already existed in PDD, and its insert position is determined by sequence number in CFG, there are differences between branching nodes and non-branching nodes, and the numbers are non-branching node and those start with P are branching nodes. Unconditional transfer statement goto is not considered in the actual process, and loop statement is considered as one loop. The sequence numbers of node satisfy two conditions: (1) if there is a path from node N_i to node N_j, the sequence number of N_j is bigger than N_i; (2) if N_i is a branching node, N_j is less than all child nodes' sequence number of N_i. In addition, when new PDD generated, if the insertion node is non-branching node, modify the set of node that has data dependent on the insertion node after it. Formal description is as follows: for new insertion node N_k, if $D(N_k) \neq \varnothing$, then for arbitrary node N_i, $N_i \in PDD \wedge i > k$, modify the set of data dependence of node N_i to $U(N_i) \cup D(N_k)$ *when* $U(N_i) \cap D(N_k) \neq \varnothing$. If it is a branching node and conflict with other node, remove all the other nodes under control of the branch node. The modify process after insert node we called modify process of influence with data dependence.

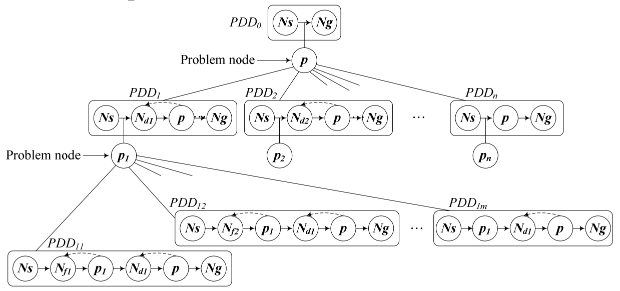

Figure 3. A search tree for PDD generation

As the program is shown in figure 1, given the starting node *1* and goal node *14*, there is PDD_0=<*1*, *14*>, generation process in the first stage is shown in table I, *PN* is the problem node generated by the searching process.

Let $PDD=PDD_{11111}$, first stage generate an incomplete execution path of program, and it cannot build the constraint system, we need to complete it by PDD generation in the second stage. Get any two adjacent nodes N_i, N_j in PDD, if $(N_i, N_j) \notin E$, searching process generate arbitrary path that satisfies the condition, inserting the nodes in the path into PDD according to insert rules, and execute modify process of influence with data dependence until the PDD is complete PDD, i.e. *PDD* = <*1, P1, P2, 3, 4, 5:{3}, P3:{4}, P4:{5}, P5:{4, 5}, P6:{4}, P7:{{4, 5}, {5}}, 14*>, get adjacent nodes *(P2, 4)* $\notin E$ and generate set of path *{<P2, 3, 4>}* from node *P2* to *4* by searching process, and select a path not conflicting with existed nodes in PDD to insert into the PDD, we insert node *3* into PDD here and execute the modification process of influence with data dependence, and get arbitrary path *{<4, 5, P3, P4, P5>, <4, 5, P3, 6, 7, 8, P4, P5>, …}* through the adjacent nodes *(4, P5)*, we insert node *5*, *P3*, *P4* into PDD and execute modify process of influence with data dependence to get the complete PDD

PDD = <1, P1, P2, 3, 4, 5:{3}, P3:{4}, P4:{5}, P5:{4, 5}, P6:{4}, P7:{{4, 5},{5}}, 14> (1)

TABLE I.
DEMONSTRATION OF PDD GENERATION IN THE FIRST STAGE

layer	PDD	PN	LD	New PDD
0	PDD0=<1,14>	P7	{4,5,7,8,10,11,ε}	PDD1,PDD2,PDD3,PDD4,PDD5,PDD6,PDD7
1	PDD1=<1,4,P7:{4},14>	P2	{ε}	PDD11
2	PDD11=<1,P2,4,P7:{4},14>	P1	{ε}	PDD111
3	PDD111=< 1,P1,P2,4,P7:{4},14>	P6	{4,5,7,10, ε}	PDD1111, PDD1112, PDD1113, …
4	PDD1111=< 1,P1,P2,4,P6:{4},P7:{4},14>	P5	{4,5,7,8,10,11,ε}	PDD11111, PDD11112, …
5	PDD11111=<1,P1,P2,4,P5:{4},P6:{4},P7 :{4}, 14>	{}	–	–

B. Search Process

Search process includes searching problem nodes and generating arbitrary path between nodes. Searching problem node is searching start node of critical branch, insert critical branch into PDD earlier in order to reduce the generation number of infeasible program path and improve the efficiency of program execution.

Search process measures all branching nodes before specify node, if there is only one path from branch node which can reach specify node, the branch is critical branch and return branching node as problem node. As searching the critical branch of specify node, descend waiting branching nodes in CFG and search the closest problem node with specify node at first. Critical branch search is achieved by the function GetProblemNode(PDD), pseudo code as follows:

algorithm GetProblemNode(PDD)
BEGIN
Copy(tail(front(PDD)), *TmpPDD*)
//Copy PDD to TmpPDD except the first
//and the last node in PDD
FOR each Node N_c in *TmpPDD*
FOR each Branch Node N_i in CFG where $N_i < N_c$ and $N_i >$ first*(PDD)*
Get N_j and N_k from CFG where $(N_i, N_j)\in E$ and $(N_i, N_k)\in E$
IF (N_j==N_c **OR** N_k==N_c)) **RETURN** N_i
min=#head(GetPathSet (N_j, N_c))
//Get arbitrary path set from N_j to N_c
max=#last(GetPathSet (N_j, N_c))
IF (*min*==2 **AND** *max*>2) **RETURN** N_i
END FOR
END FOR
RETURN NULL
END GetProblemNode

Algorithm measures all branching nodes before specify node, if there is only one path from branch node which can reach specify node, the branch is critical branch and return branching node as problem node. As searching the critical branch of specify node, descend waiting branching nodes in CFG and search the closest problem node with specify node at first, i.e. in *PDD=<3, P6, 14, 15>*, all *(P5, P6)*, *(P6, P7)* and *(P7, 14)* are critical paths for node *14*, and return the node *P7*.

C. Constraint System Building

PDD is the program execution node sequence, and N:LDP(N) contains the node number and the last definition node, i.e. formula (1), there are 12 nodes including mutation node, and 12 sets of data dependence. We cut CFG by PDD, remove the node not in PDD and add the back edge with data dependence, Demonstration graph of CFG split by formula (1) is shown in figure 4.

(1)
(2)

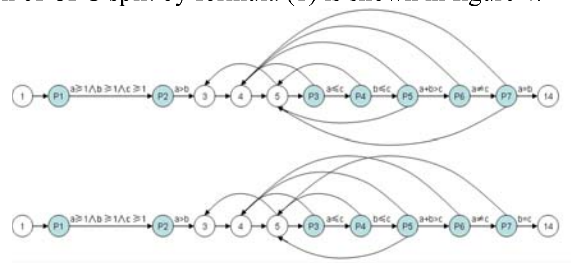

Figure 4. Demonstration graph of CFG split by PDD

The PDD is the path with data dependence from start node N_s to goal node N_g. C_n is necessity condition, P is a sequence of paths built by reachability condition, initialize to <>, N points to the node of PDD, C is predicate expression of reachability condition generated by P, initialize to "". The algorithm can be divided into two parts: (1) clear data dependence line in cutting CFG; (2) clear control line, which generates constraint expression C by P, build predicate expression of constraint system by combine C and necessity condition.

Building constraint system by formula (1), after step 1 is as follow: there are PDD = <14>, P = <1, P1, P2, 3, 4, 5', P3, P4, P5, P6, P7, 14>, change 5' to b = a, branch predicates are BP(P3, P4) = "b <= c", BP(P4, P5) = "a <= c", BP(P5, P6)= "b+a > c", BP(P6, P7) = "b != c", BP(P7, 14) = "b = a ∨ a = c". Use mutation operator SCR on node 14, there is 14: Δtype = type, necessity condition is C_n = ((type = "ISOSCELES") != (Δtype = type)), LDP(14) = {1}, substitute node 1 to necessity condition, there is C_n = ((type = "ISOSCELES") != (type = "SCALENE")) = true. In step 2, execute the branch predicates of the path P, return value as follows:

"(a >= 1 ∧ b >= 1 ∧ c >= 1) ∧ (a > b) ∧ (b <= c) ∧ (a <= c) ∧ (b + a > c) ∧ (b != c) ∧ (b = a ∨ a = c) ∧ (true)".

Then split Disjunction expression by permutation and combination, and get conjunction expression as following:

"(a >= 1 $\wedge$ b >= 1 $\wedge$ c >= 1) $\wedge$ (a > b) $\wedge$ (b <= c) $\wedge$ (a <= c) $\wedge$ (b + a > c) $\wedge$ (b != c) $\wedge$ (b = a) $\wedge$ (true)". (1)’

"(a >= 1 $\wedge$ b >= 1 $\wedge$ c >= 1) $\wedge$ (a > b) $\wedge$ (b <= c) $\wedge$ (a <= c) $\wedge$ (b + a > c) $\wedge$ (b != c) $\wedge$ (a = c) $\wedge$ (true)". (1)’’

D. Auto-detection of Equivalent Mutants

If the mutant is equivalent mutant, it will always get the same output as the original program, so the test case cannot kill it. If we can automatically detect the equivalent mutants, we can save much TDG time and improve the TDG efficiency. Strategies for Detecting Equivalent Mutants are as follows:

1. Negation [26]

1) **Negation**, Constraint C_1 is the negation of C_2 iff they describe non-overlapping domains and cover the entire domain.

2) **Partial Negation**, Constraint C_1 is a partial negation of C_2 if they describe non-overlapping domains and not cover the entire domain.

3) **Semantically Equal**, Two constraints C_1 and C_2 are semantically equal if they describe the same domain.

4) **Syntactically Equal**, Two constraints C_1 and C_2 are syntactically equal if they describe the same domain and same string of symbols.

2. Constraint Splitting [26]

Give two constrains (x + y) > 0 and (x < 0) $\wedge$ (y < 0). The negation strategy cannot get the two constraints conflict, constraint splitting is used to detect this conflict. Give two constraints C_1 and C_2. Then give two new constraints C_3 and C_4 that $C_1 \Rightarrow C_3 \vee C_4$. If both C_3 and C_4 conflict with C_2, we can get C_1 conflicts with C_2.

3. Constants Comparison [26]

Let A be the constraint (X rop1 K_1) and B be the constraint (X rop2 K_2) where X is a variable, rop1 and rop2 are relational operators, and K_1 and K_2 are constants, we can often decide whether A conflicts with B by evaluating the two constants and relational operators. If we use the negation strategy to constraints (x > 1) and (x < 0), it can partially negate or negate to (x < 1) or (x ≤ 1), but neither (x < 1) nor (x ≤ 1) is syntactically equal to (x < 0), so we cannot get they conflict. For this problem, constants comparison can be used to get the two constraints conflict.

There is p’ =(a>=1 $\wedge$ b>=1 $\wedge$ c>=1) $\wedge$ (a>b) $\wedge$ (b<=c) $\wedge$ (a<=c) $\wedge$ (b+a>c) $\wedge$ (b!=c) $\wedge$ (b=a) in formula (1)’. Using detecting equivalent mutants strategy 1 negation strategy, (a>b) and (b=a) are conflicted with each other, and there are equivalent mutants or the path is infeasible path. Return to formula (1)’’, p’’ =(a>=1 $\wedge$ b>=1 $\wedge$ c>=1) $\wedge$ (a>b) $\wedge$ (b<=c) $\wedge$ (a<=c) $\wedge$ (b+a>c) $\wedge$ (b!=c) $\wedge$ (a=c), there is no conflict after using each detecting equivalent mutants strategy, go on executing the solving process.

E. Constraint System Solving

Constraint system solving process is a process of operate domain reduction indefinite form. First, initial the program input. Meanwhile it saves variable and its initial domain into DDS.

Initialize the DDS for a, b and c in formula (1)’’ p’’ = (a >= 1 $\wedge$ b >= 1 $\wedge$ c >= 1) $\wedge$ (a > b) $\wedge$ (b <= c) $\wedge$ (a <= c) $\wedge$ (b + a > c) $\wedge$ (b != c) $\wedge$ (a = c) as follow: a | -100..100; b | -100..100; c | -100..100. Domain reduce to variable a firstly according to a>=1. There are ldomain = -100 .. 100, const = 1 in function ExprDomain, and DDS = a | 1 .. 100; b | -100 .. 100; c | -100 .. 100 executed by function update with domain reduce to variable a. In the same domain reduce to b >= 1 and c >= 1, there is DDS = a | 1 .. 100; b | 1 .. 100; c | 1 .. 100.

Domain reduce to variables a and b according to a > b, and calculate ldomain | 1 .. 100, rdomain | 1 .. 100 by function ExprDomain, the left and right domain satisfy the condition 1 of GetSplit, (ldomain.Bot ≥ rdomain.Bot) and (ldomain.Top ≤ rdomain.Top), the first split parameter is I = 1/2, calculate the split point is SplitPoint = (ldomain.Top – ldomain.Bot) × i + ldomain.Bot = (100 – 1) × 1/2 + 1 = 50. Then domain reduce to variables a and b by function update, there is DDS = a | 51 .. 100; b | 1 .. 50; c | 1 .. 100.

The same domain reduce operates to variables b and c according to b <= c, there is DDS = a | 51 .. 100; b | 1 .. 25; c | 25 .. 100. According to a <= c, there is DDS = a | 51 .. 75; b | 1 .. 50; c | 75 .. 100. Domain reduce to variables a, b and c according to (b + a > c), there is DDS = a | 69 .. 75; b | 19 .. 50; c | 75 .. 87. Domain reduce to variables b and c according to b != c, there is no split condition for function GetSplit, return TRUE, there is DDS = a | 69 .. 75; b | 19 .. 50; c | 75 .. 87. Domain reduce to variables a and c according to a=c, there is DDS = a | 75; b | 19 .. 50; c | 75. Chooses random variables in domain of a, b and c, and solves the constraint system by back substitution, i.e. a = 75, b = 23, c = 75.

TABLE II.
PROCESS OF SOLVING THE CONSTRAINT SYSTEM FOR FORMULA (1)’’

Step	GetSplit Condition	SplitPoint	a	b	c
1.Start	-	-	-100 ..100	-100 ..100	-100 ..100
2.a>=1	rexpr is a constraint	-	1.. 100	-100 ..100	-100 ..100
3.b>=1	rexpr is a constraint	-	1.. 100	1.. 100	-100 ..100
4.c>=1	rexpr is a constraint	-	1.. 100	1.. 100	1.. 100
5.a>b	(ldomain.Bot>=rdomain.Bot) and (ldomain.Top<=rdomain.Top)	(100-1) /2+1	51.. 100	1..50	1.. 100
6.b<=c	(ldomain.Bot>=rdomain.Bot) and (ldomain.Top<=rdomain.Top)	(50-1) /2+1	51.. 100	1..25	25.. 100
7.a<=c	(ldomain.Bot>=rdomain.Bot) and (ldomain.Top<=rdomain.Top)	(100-51) /2+51	51.. 75	1..25	75.. 100
8.b+a>c	(ldomain.Bot>=rdomain.Bot) and (ldomain.Top<=rdomain.Top)	(100-75) /2+75	69.. 75	19.. 25	75.. 87
9.b!=c			69.. 75	19.. 25	75.. 87
10.a=c			75	19.. 25	75

V. EXPERIMENTAL ANALYSIS

Experimental environment: CPU is Inter E7400, CPU frequency is 2.80GHz, memory size is 4G, hard disk size is 160G and OS is Ubuntu 10. CBT is assembled in tool Godzilla [3], CAT uses tool TESTGEN [1] and DRD is a small TDG tool implemented by Java. The experimental program is shown in table III. All the programs we used to experiment are simple structures, and these programs have distinct characteristics.

TABLE III.
EXPERIMENT PROGRAMS

Program	Description	Statements	Mutants
Mid [15]	Return the middle value of three given integers	11	26
TriType	Classify the triangle as equilateral, isosceles, scalene or invalid	22	44
MinMax	Return the maximum and minimum elements from an integer array	10	23
GCD	Return the greatest common divisor of two given positive integers	9	19
Sample [18]	A sample function used in paper [18]	17	30

We use the CBT, CAT and DRD with detecting equivalent mutants to generate test data and the assess value for experimental are as follow:

(1) Success ratio of test data generation, which is the same to mutation score. We set the run time threshold to 100s. If the valid test data is generated in run time threshold, we successfully generate the test data. We use MS to record the success ratio as assess its reliability.

(2) Average time of successfully generate the test data marked as ASuc and maximum time of successfully generate the test data marked as MSuc to assess execution efficiency.

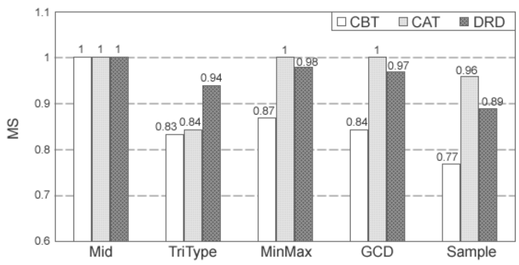

Figure 5. Comparison diagram of mutation score

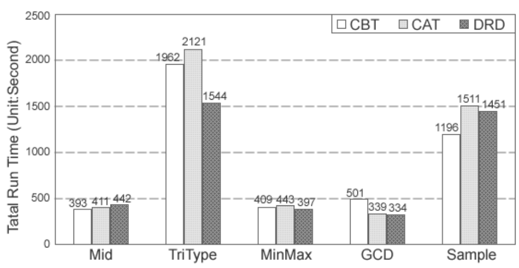

Figure 6. Comparison diagram of total run time

We use CBT combined with DDR, CAT and DRD with detecting equivalent mutants to those programs to generate the test data. The result of MS is shown in figure 5. DRD is higher than CBT in MS, but lower than CAT except for TriType. CAT is Goal-oriented TDG method, so CAT always has a high MS.

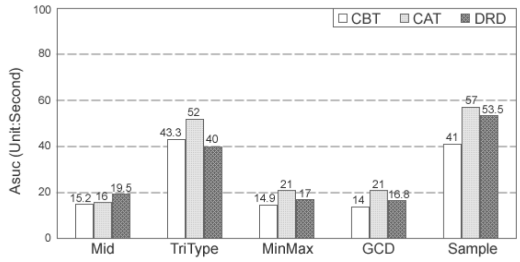

Figure 7. Comparison diagram of ASuc

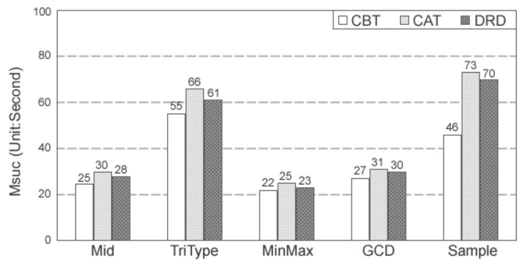

Figure 8. Comparison diagram of MSuc

Although DRD has less MS than CAT, but DRD reduce 3%~8% runtime than CAT as shown in figure 6. If there are more branch statements, DRD will take less run time than CBT. So we can get the conclusion that DRD has higher MS and reduces the total run time.

The results of ASuc and MSuc are shown in figure 7 and 8. CBT has the fastest TDG and lower run efficiency. Although CAT has a highest MS, while CAT has highest ASuc and MSuc. DRD combined the advantage of CBT and CAT, DRD has lower ASuc and MSuc than CAT, and it also get higher MS than CBT.

MS is conflict with run time. High MS always spend more run time. Experiment results show that DRD has better execution efficiency than CBT and CAT. It can successfully judge the equivalent mutants and decrease the possibility to select the infeasible path. It also can solved influences on constraint systems caused by the multiple conditional branch statement.

VI. CONCLUSION

Mutation testing has been developed for 30 years but remains active. It is not only applied in unit tests but also contributes to a lot of theoretical research in interface testing, aspect-oriented testing, object-oriented testing and contract testing. Mutation testing is a very labor-intensive process, which spends most resource in TDG. Automatically TDG process can greatly improve the efficiency of software testing; thereby reduce the cost of software development. On the one hand, Mutation testing is a testing strategy to evaluate the completeness of test cases, which helps the testers to control the quality of tests and builds confidence for software accuracy. On the other hand, mutation testing can automatically generate complete test cases, which greatly improved the quality and efficiency of software testing. This paper proposed a new method DRD combined with goal-oriented method for TDG with data dependence. It added the detecting of equivalent mutants and solved the problem of branch statement composed by many conditions. It builds constraint system by control flow analysis, data flow analysis, symbols execution and mutation information, used domain reduction technology and solved the constraint system by selecting random minimum variables in selected domains and verified with back substitution. It adds the detecting of equivalent mutants and solves the effect on the constraint systems caused by branch statement, which is composed by multiple conditions. Experimental results showed that this method improved the success rate and execution efficiency of test data generation to a large extent though further improvement in total run time is needed. In this paper, data dependence has been considered in PDD generation process; it insert the problem nodes and those has data dependence on them into the search tree firstly, but in the experiment of program TriType, it is always faster to find solutions in constraint system building without data dependence. The next step is to consider how to evaluate the insertion of data dependence according to the program.

REFERENCES

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” IEEE Computer, vol. 6, no. 4, pp. 34-41, 1978.

[2] F. G. Hamlet, “Testing programs with the aid of a compiler,” IEEE Transactions on Software Engineering, vol. 3, no. 4, pp. 279-290, 1977.

[3] A. J. Offutt, “Investigations of the software testing coupling effect,” ACM Transactions on Software Engineering Methodology, pp. 3-18, January 1992.

[4] R. A. DeMillo, A. J. Offutt, “Constraint-based automatic test data generation,” IEEE Transactions on Software Engineering, vol. 17, no. 9, pp. 900-910, 1991.

[5] Y. S. Ma, Y. R. Kwon, and J. Offutt, “Inter-Class Mutation Operators for Java,” Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE 2002), Annapolis, MA, USA, pp. 352-363, 2002.

[6] H. J. Lee, Y. S. Ma, and Y. R. Kwon, “Empirical Evaluation of Orthogonality of Class Mutation Operators,” Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004), Washington, DC, USA:IEEE Computer Society, pp. 512-518, 2004.

[7] A. S. Namin, J. H. Andrews, “Finding Sufficient Mutation Operators via Variable Reduction,” Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)(MUTATION'06), Leipzig, Germany, pp. 351-360, 2006.

[8] B. Korel, A. M. Al-Yami, “Assertion-oriented automated test data generation,” Proceedings of the 18th international conference on Software engineering, Berlin, Germany, pp. 71-80, 1996.

[9] M. E. Delamaro, J. C. Maidonado and A. P. Mathur, “Interface mutation: An approach for integration testing,” IEEE Transactions on Software Engineering, vol. 27, no. 3, pp. 228-247, 2001.

[10] L. J. Morell, “A theory of fault-based testing,” IEEE Transactions on Software Engineering, vol. 16, no. 8, pp. 844-857, 1990.

[11] X Liu, G Xu, X Fu, Y Dong, “Test Data Generation Considering Data Dependence,” 2010 Fifth International Conference on Frontier of Computer Science and Technology, pp. 208-213, 2010.

[12] Z Qian, “Test Case Generation and Optimization for User Session-based Web Application Testing,” Journal of Computers, vol. 5, no. 11, pp. 1655-1662, 2010.

[13] C Wang, J HeHe, G Li, J Han, „An Automated Test System for Flight Simulator Fidelity Evaluation,” Journal of Computers, vol. 4, no. 1, pp. 1083-1090, 2009.

[14] Jon Edvardsson, “A survey on automatic test data generation,” Proceedings of the second conference on computer science and engineering, pp. 21-28, 1999.

[15] A. J. Offutt, Z. Jin and J. Pan, “The dynamic domain reduction procedure for test data generation,” Software: Practice and Experience, vol. 17, no. 9, pp. 900-910, 1991.

[16] J. H. Shan, Y. F. Gao, M. H. Liu etc, “A new approach to automated test data generation in mutation testing,” Chinese journal of computer, vol. 31, no. 6, pp. 1025-1034, 2008.

[17] AM Alakeel, “An Algorithm for Efficient Assertions-Based Test Data Generation,” Journal of Software, vol. 5, no. 6, pp. 644-653, 2010.

[18] R. Ferguson, B. Korel, “The chaining approach for software test data generation,” ACM Trans on Software Engineering and Methodology, vol. 5, no. 1, pp. 63-86, 1996.

[19] N. Gupta, A.P. Mathur and M.L. Soffa, “Automated test data generation using an iterative relaxation method,” Proceedings of the ACM SIGSOFT Sixth international symposium on Foundations of software engineering, pp. 231-244, 1998.

[20] S Wang, Y ji, S Yang, “A Micro-Kernel Test Engine for Automatic Test System,” Journal of Computers, vol. 6, no. 1, pp. 3-10, 2011.

[21] K Zhou, X Wang, G Hou, J Wang, S Ai, “Software Reliability Test Based on Markov Usage Model,” Journal of Software, vol. 7, no. 9, pp. 2061-2068, 2012.

[22] K Li, Z Zhang, J Kou, “Breeding Software Test Data with Genetic-Particle Swarm Mixed Algorithm,” Journal of Computers, vol. 5, no. 2, pp. 258-265, 2010.

[23] Ahmed S. Ghiduk, “Automatic Generation of Object-Oriented Tests with a Multistage-Based Genetic Algorithm,” Journal of Computers, vol. 5, no. 10, pp. 1560-1569, 2010.

[24] Xie X Y, Xu B W, Shi L, Nie C H, “Genetic test case generation for path-oriented testing,” Journal of Software, vol. 20, no. 12, pp. 3117-3136, 2009.

[25] Dhatchayni M, Arockia Xavier Annie R, Yogesh P, Benet Zacharias, “Test Case Generation and Reusing Test Cases for GUI Designed with HTML,” Journal of Software, vol. 7, no. 10, pp. 2269-2277, 2012.

[26] A. J. Offutt, J. Pan, “Automatically detecting equivalent mutants and infeasible paths,” Software testing, verification and reliability, pp. 165-192, 1997.

[27] Miu WC, Li G, Zhu GM, “Software engineer language–Z,” ShangHai scientific and technology literature publishing house, 1999