Polynomial Constraint Satisfaction Utilizing Adaptive Dynamic Domain Reduction

Sajjad Naghdali Zanjani[1,2]
Computer Engineering department
[1]Faraniroo Engineering Co.
[2]Islamic Azad University Qazvin branch
Tehran, Iran
sazanj@hotmail.com

Mehdi Dehghan Takht fooladi
Computer Engineering department
Amirkabir University
Tehran, Iran
dehghan@aut.ac.ir

***Abstract*—Automation of generating test data is an adequate solution for testing software. Moreover, solving constraints is one of the most important parts of this process. Each constraint is a condition along which there is a vector with different variables' values involves in condition to satisfy the constraint. Despite the fact that previous methods have investigated the solver vectors by decreasing the value's domain, they do not have the capability of finding all solver vectors. This paper proposes Adaptive Dynamic Domain Reduction, which is referred to as ADDR, in order to find all solver vectors related to the polynomial constraints. In the proposed method, each constraint is satisfied utilizing domain reduction or a divide-and-conquer based method. In iterations of algorithm, new domains are defined for variables of a constraint with the results that choosing each arbitrary member of the domain can satisfy constraint. Moreover, union of these domains can find all solver vectors. In order to demonstrate the efficiency of the proposed method, a set of experiments is conducted. These experiments show the proposed method can further improve the process of test data generation in a situation in which complicated or simultaneous constraints may result in the automation failure.**

Keywords- test data generation, constraint solving, split point, feasible area.

I. INTRODUCTION

Software testing is a fundamental part of software production life cycle, although it is time consuming and requires big budget and labor intensive. It was observed in the course of software production that half of budget is associated with testing [1]. Furthermore, it is definitely proved that in almost all instances automation of test data generation can decrease the budget of software production [2]. The procedure of generating test data aims to determine the software inputs with the result that test criteria will be satisfied. Different criteria have been proposed for test data generation in which each criterion represents a specific method. As an example of test data generation criteria, it can be referred to the statement/branch coverage in path wise tests [3, 12, 5] or kills a mutant method in mutation tests [6, 7].

Generating test data methods can be classified into three classes as following below:

1. *Randomized generation:* This simple method utilizes randomized algorithms to generate test data and does not need to review the source code. Indeed, the test data, which is generated in this manner, are not chosen based on the source code and lack the ability of highlighting small semantic errors.
2. *Goal oriented generation:* In this method, test data is generated in a way that all program's paths are covered. As this method covers an unspecific path in each initiation, it does not face infeasible path [8].
3. *Path oriented generation:* This method is the strongest one in which a specific path is chosen at first. Then, it endeavors to generate the test data in a way that a specific path is traversed [4].

All test data generation methods, excluding randomized method, have to satisfy several constraints. Demilli and Offutt [6] develop a technique, so-called Constraint Based Testing (CBT), in order to solve constraints. However, CBT does not work well when several constraints should be satisfied in conjunction with each other. Additionally, Offutt et al. [5] improves CBT and proposes an algorithm called Dynamic Domain Reduction (DDR). Even though DDR covers a considerable number of CBT's defects, it encounters problems in solving complicated constraints. Subsequent to DDR, much research has been devoted to solve the problems of DDR in general form [7, 8]; nonetheless, these optimizations are not relative to solve constraints.

This paper contributes towards DDR algorithm and proposes an adaptive dynamic domain reduction algorithm in order to solve DDR's problem and to satisfy polynomial constraints, which are more complicated. This algorithm, hereinafter referred to as ADDR, has the capability of finding nearly all solver vectors and increasing the chance of incoming constraints satisfaction. In the proposed method, first, a set of simple and fundamental constraints is defined, which are solved based on domain splitting. Then, each polynomial's constraint is converted to these constraints by means of variable substitution. In each ADDR iteration, new domains are defined for the variables of a constraint with the results of choosing each arbitrary member of the domain having the capability of satisfying constraint. Finally, this set of domains will investigate all the solver vectors and increase the chance of satisfying incoming conditions. Therefore, our focus in this research is not only on satisfying constraints by utilizing divide-and-conquer method, but also on finding all solvers vector.

The roadmap of this paper is as follows: Preliminary information and background are presented in Section II. Section III dedicates to the CBT and DDR definition and their limitations. ADDR is expressed in detail in Section IV. Experimental results are presented in SectionV and finally, this paper is concluded in SectionVI.

II. PRELIMINARY DEFINITIONS

Before we continue our discussion about the ADDR, we should express and clarify several preliminary definitions, which are control flow graph, constraint, constraint structure, constraint satisfaction, constraint evaluation, feasible area, complete solution of constraint, and test data generation.

A. *Control Flow Graph*

This directed graph, so-called CFG, depicts the structure of program and data flows. Each node in the graph is a condition or a basic block node. A complete path in this graph consists of nodes which start at the entry node and end up at the exit node.

B. *Constraint*

All condition nodes of CFG introduce a constraint. Constraint is an algebraic expression which introduces a condition and limits the variable domain of the condition. For instance, the A>0 limits the domain of variable A to the positive values. It is worth pointing out in this context that we use expression or condition term instead of constraint.

C. *constraint Structure*

As mentioned above, a constraint is an algebraic expression consists of input or local variables which are in relation with each other by a conditional operator {=, >, <, ≥, ≤} and its output is True or False. Furthermore, it is possible to restate constraints based on input variables through a symbolic execution [9, 10].

In this paper, "exp ∂ const" form is utilized for expressing constraints in order to simplify all expressions. Where, "exp" is an algebraic expression, ∂ is a conditional operator {=, >, ≥}, and "const" is a constant. It is worth pointing out in this context that all expressions can be restated with ∂.

D. *Constraint Satisfacion*

A constraint can be defined as a function $C: S \rightarrow R$, where S is all possible input values of constraint's variable and R is the range of C that is selected from {*True, False*} set. Precisely, S is the set of all vectors $V = (value_1, value_2, \ldots, value_n)$, where $value_i \in D_{xi}$ and D_{xi} is the domain of variable x_i. If the output of function C is true then the constraint is satisfied.

E. *Constraint Evaluation*

In the event that the constraint C is evaluated by vector "V", the output is true or false. For instance the expression $3x\text{-}2y>3$ is not solved by $V = (1, 0)$, this is because $3>3$ is a false condition's result. In order to test data generation process for a specific path, this data should have the capability of satisfying constraints in the path.

F. *Feasible Area*

If a constraint is illustrated on a Cartesian plane, an area may exist where the constraint will be satisfied. This area is called feasible area and is limited by the initial domain of variables involved in the condition. It is possible that there is no feasible area; therefore, no vector can solve constraints.

G. *Complete Solution of Constraint*

If all solver vectors are obtained, then it can be definitely claimed that the constraint is completely solved. In other words, complete solving of a constraint, which we referred to as a solving constraint, equals to cover the feasible area completely by means of introducing variables' domain.

H. *Test Data Generation*

There are paths $p_1, p_2, \ldots, p_n$ in the CFG, which all test data generation methods (excluding randomized method) aim to find program's inputs with the result that maximum number of these paths are traversed [4,11]. Each p_i consists of m constraints $C_{i1}, C_{i2}, \ldots, C_{im}$. In order to traverse p_i, the input data must be chosen in a way that the output of $C_{i1} \wedge C_{i2} \wedge \ldots \wedge C_{im}$ has a true value, i.e., all C_{ij} should be satisfied. Therefore, with improving the constraints satisfaction and solver process, the procedure of test data generation will be improved.

III. CBT AND DDR IN DETAIL

Demilli and Offutt develop a technique, so-called Constraint Based Testing (CBT), to solve constraints where a CFG, constrains satisfaction, and symbolic execution are utilized to generate test data. In CBT, generated test data must satisfy reachability, necessity, and sufficiency conditions of each mutant. In fact, input data must have the capability of visiting the mutated statement (reachability condition). Moreover, after executing mutated statement, the error must be detected (necessity condition) and propagated until the output generates (sufficiency condition).

CBT satisfies constraints by means of domain reduction. To achieve this, first, a domain is defined for each constraint's variable. This domain is defined by a tester or is derived from properties and preconditions of the application. Then, with respect to condition of constraint, the domain is reduced. More precisely, suppose that we have constraint in the form of $x \partial \omega$ where x is a variable, ∂ is a relational operator, and ω is a variable or constant. If ω be a constant, the domain of x is just reduced, else domain of x and ω will be reduced together. In a situation in which we have complicated constraint, we replace this constraint with a selective value of variable in order to solve it desirably. When there is no way to simplify the constraint, a value from one of constraints' variable is selected based on a heuristic method and replace in all constraints. This procedure continues until all variables get this value.

CBT method suffers from two major problems which are (1) It utilizes a heuristic method for selecting values of variables where the probability of failure for selecting value of other variables dramatically increases. Moreover, it is possible that one value is chosen more than once. (2) This method lacks the capability of solving complicated expressions in constraints. Therefore, it may ignore the constraints or break them into approximated simple forms.

DDR is proposed by Offutt et al. to solve problems of CBT; however it could achieve more success in solving the first problem than the second one. DDR combines three methods including CBT, Korel dynamic method [11], and symbolic execution to domain reduction. The inputs of this algorithm are CFG, graph's input and output node, and initial domain for all input variables. At the first step, a set of finite complete paths is selected. Then, each path is examined in detail sequentially and its constraints are restated based upon the input variables through symbolic execution.

A simple constraint can be represented by $x \partial \omega$, where ω is a variable or constant. DDR method utilized GetSplit function to solve this constraint. In a situation in which the algorithm faces complex constraints it applies divide and conquer method to constraints with the aim of making them simple. GetSplit function gets initial variable domain and search index, where this index shows the kth try for satisfying constraint, as its input. Then, by utilizing these inputs reduces the domain of two variables with the result that any value of new domain can satisfy the constraint. In this function, first, a point so-called split point is selected, which is common in both domains. Then, these domains are split with the result that there is no overlap. Each member of new domains has the capability of satisfying all constraints.

In a number of cases, selected split point has not the capability of satisfying incoming condition. In this situation a new split point is introduced based on a recursive method. Although several hints are considered in DDR algorithm for dealing with polynomial expressions, it lacks the ability of detecting all solver vectors and fails to satisfy complex constraints. With the aim of solving complex constraints in the form of polynomial expression, an adaptive method towards DDR is proposed, which will be explained it in following section.

IV. Proposed Method

Adaptive Dynamic Domain Reduction is an adaptive extension towards DDR method which is based on the divide and conquer method. ADDR has the capability of finding all solver vectors of a constraint in the form of *exp>const* or *exp≥const.* Here, *exp* is a polynomial expression where each term consists of variable with different power and *const* is a constant. We model our definition as follows below (1):

$$\begin{array}{l} \boldsymbol{exp > const} \quad \textbf{or} \quad \boldsymbol{exp \geq const} \\ s.t. \\ exp = term_1 \pm term_2 \pm \ldots \pm term_n \\ term_i = \pm\, x_{i1}^{Pi1} (\times/\div)\, x_{i2}^{Pi2} (\times/\div)\, x_{i3}^{Pi3} \ldots (\times/\div)\, x_{in}^{Pin} \end{array} \qquad (1)$$

It is clear that the divide and multiply can use in behalf of each other. However, in (1) we separate them for the reason of divide by zero. In the proposed method, just five fundamental constraints are directly solved and other complex constraints will be transformed to these fundamental constraints. These fundamental constraints are as follows (2):

1. $c\times A>const$ **or** $c\times A=const$
2. $A-B>0$
3. $\pm c\times A\times B>const$ (2)
4. $\pm c\times A\div B>const$
5. $A^P>const$ **or** $A^P<const$

Where c and p are positive values and A and B are two variables which their initial domains has been defined. These domains are defined by a tester or are derived from properties and preconditions of the application. It should be noted that we could utilize ≥ instead of > in (2).

In solving fundamental expression, a split point is utilized along which its regulation introduces new domains and the set of all domains can fairly cover all solver vectors. As solving all constraints is associated with solving fundamental constraints, if all fundamental constrains are solved, then all solver vectors will be found for all constraints. In fact, the satisfactory vectors of a path are found if and only if the path is feasible.

We divide our discussion into two paradigms including (1) solving the fundamental constraints and (2) solving complex constraints. Without loss of generality, it is assumed that a constraint consists of two variables A and B which their domains are $D_A= (A.b, A.t)$ and $D_B = (B.b, B.t)$, respectively.

A. Satisfying Fundamental Constraints

In order to solve fundamental constraints (excluding 2.1 and 2.5), a point called split point is defined which is used to divide the variable domain into two parts.

1) Satisfying $A^P>const$ and $A^P<const$.

These constraints can be formulated as follow:

1. $A^p>const$, if $const>0$
2. $A^p>const$, if $const<0$ and $p=2k+1$
3. $A^p>const$, if $const<0$ and $p=2k$
4. $A^p<const$, if $const>0$
5. $A^p<const$, if $const<0$ and $p=2k+1$
6. $A^p<const$ if $const<0$ and $p=2k$

where k be a positive value, and p is a positive integer expression. In conditions 1 and 2, the new domain is $D_A= (const^{(1/p)}, A.t)$. Condition 3 is always true and domain of A

does not change. Same as former condition, condition 4 and 5 limit the domain of A as D_A = ($A.b$, $const^{(1/p)}$) and finally, condition 6 is always false and the domain of A is null.

2) Satisfying ±c×A > const

This expression can be restated as $-A>(const \div c)$ or $A>(const \div c)$. In order to satisfy this constraint, the domain of variable, A, should be reduced as D_A= ($const \div c_1$, $A.t$) or D_A= ($A.b$, $-const \div c$), in turn. If the constraint defines as $\pm c \times A = const$, then the variable A can satisfy the constraint with the $\pm const \div c$ value.

3) Satisfying A-B > 0

In this constraint, on those occasions in which two domains do not have any intersection then the condition is true if $A.b>B.t$ and the condition is false if $B.b>A.t$. In contrast, when two domains have intersection the split point is utilized. This point reduces two domains with the result that D_A= *(splitPoint, A.t)* and D_B= *(B.b, splitPoint)*. The value of split point completely depends on the number of iterations, so-called search index (*srchIndex*), in order to satisfy constraint. The following procedure shows how a split point can be computed:

$$srchpt = \frac{2(srchIndex - 2^{exp}) + 1}{2^{(exp+1)}} \qquad bottom = max\ (A.b,\ B.b)$$
$$exp = \lfloor log_2\ srchIndex \rfloor \qquad top = min\ (A.t,\ B.t\ )$$
$$splitPoint = bottom + srchpt \times (top\text{-}Bottom)$$

where *srchpt* shows the direction of moving split point towards the (*bottom, top*). In order to clarify this method, we concentrate our discussion, without loss of generality, on a simple form as shown in Fig. 1.

As evident from Figure 1, striped area indicates the feasible area; furthermore, there are two bold lines on A and B basis vectors which show initial domains. In order to specify the span of split point, the $\overrightarrow{ef}$ vector is projected on B vector and in a situation in which the split point moves, then a new coordinate *(splitPoint, splitPoint)* will be selected on $\overrightarrow{ef}$ vector, which is called *sp*.

The first *sp* is placed in the middle of $\overrightarrow{ef}$ vector and with changing coordinate of split point, new domains will be found for A and B variables. Here, we show these domains by means of rectangles that their length is *A.t-splitPoint*, their width is *splitPoint-B.b*, and their left-bottom corner is placed in (*splitPoint, B.b*). In a case in which the integration of all obtained rectangles would be equal to the feasible area, the feasible area is covered. Fig. 2 (a) and (b) clarifies this matter for the first and second iterations.

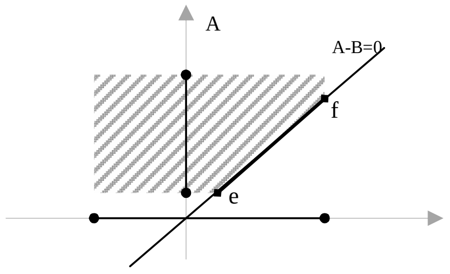
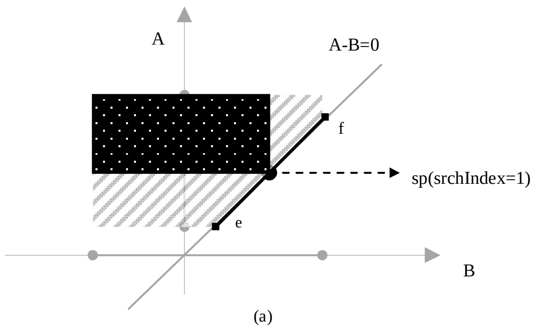


Figure 1. Feasible area related to the constraint A-B>0



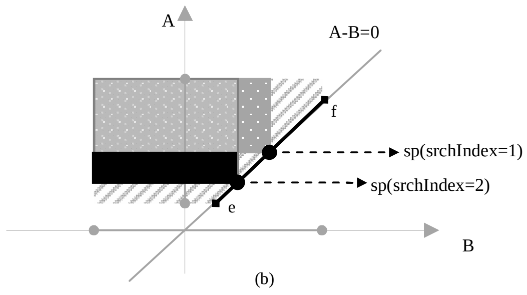


Figure 2. Covering feasible area by changing split point

The dark rectangle in Fig. 2 (b) depicts the covered perimeter in the second iteration along which $A\text{-}B>0$ is satisfied. In each iteration, a part of feasible area is covered and in a case that the value of *srchIndex* tends toward infinity, the probability of covering feasible area tends to one. The supreme value of search Index is obtained subjectively, which will be explained in section V.

4) Satisfying ±c×A×B>const

In order to satisfy this constraint, it should be restated as a simple form of $\pm A \times B > cv$, where $cv = const \div c$. This expression can be examined in five cases. In the fifth case, the domain of B is negative ($D_B<0$) and in the other cases the domain of B is positive ($D_B>0$). The first and second cases are solved straightforwardly and other cases must be restated based on the first and second cases. In a situation in which the domain of B contains both positive and negative values, the constraint is solved in the positive domain of B, at first. If it lacks the ability to achieve the result, then it will be solved in the negative domain of B. These cases are as follows:

1- If $cv<0$ and the constraint be $-A \times B > cv$, then it can be restated as $A \times B < |cv|$. It should be noted that this condition establishes in the positive domain of B. In order to achieve new domains for A and B in solving $A \times B < |cv|$ a split point is utilized as follows:

$$D_A = (A.b,\ |cv| \div splitPoint),$$
$$D_B = (B.b,\ splitPoint)$$
$$splitPoint = B.b + srchpt \times (End\text{-}B.b) \qquad (3)$$
$$End = min\{cv \div A.b,\ B.t\} \quad if \quad A.b>0$$
$$End = B.t \quad if \quad A.b<=0$$

The *value* of split point moves across the vector *B* and is in the range of (*B.b, End*). In each step that *sp* is calculated, it is possible to illustrate it on the $A\times B=cv$ curve with coordinate of (*splitPoint*, $cv\div splitPoint$). The feasible area related to this constraint and two calculated split points are shown in Fig. 3. As it is obvious, new domains are introduced by calculating new split points with the result that feasible area will be fairly covered.

2- If cv>0 , DB>0, and the constraint is A×B>cv, then the new domains for A and B are defined as below (4):

$$D_A = (cv\div splitPoint, A.t),\ D_B = (splitPoint, B.t) \quad (4)$$

3- If $cv>0$, $D_B>0$, and the constraint is $-A\times B>cv$, then a new variable is utilized instead of *A*, $A=-A$\` in order to change $-A\times B>cv$ into $A\text{\`}\times B>cv$. Then the initial domain of A\` is calculated by domain of *A* as $D_{A\text{\`}} = (-D_A.t, -D_A.b)$. Afterwards, the $A\text{\`}\times B>cv$ is solved same as the condition 1 and the new domain of *A* is calculated based on the new domain of *A*\`. The new domain of A is $D_A = (-newD_{A\text{\`}}.t, -newD_{A\text{\`}}.b)$, where $newD_{A\text{\`}}$ stands for new domain of *A*\`.

4- If $cv<0$, $D_B>0$, and the constraint is $A\times B>cv$, then a new variable is utilized instead of *A*, $A=-A$\`. After calculating the initial domain of A\` and solving constraint based on the procedure which introduced in condition 2, the new domain of A is obtained as $D_A = (-newD_{A\text{\`}}.t, -newD_{A\text{\`}}.b)$.

In the mentioned conditions the domain of *B* contains positive values; however, it is possible that $D_B<0$. In this situation, a new variable is utilized instead of *B*, $B=-B$\`. Afterwards, the initial domain of B\` should be calculated and then the constraint can be solved based on the proposed solution in the conditions 1 to 4 and the new domain for *B* is $D_B = (-newD_{B\text{\`}}.t, -newD_{B\text{\`}}.b)$.

5) *Satisfying* $\pm c\times A\div B>const$

With the purpose of solving this constraint, firstly, it is rewritten as a simple form of $\pm A\div B>cv$. Then, two sides of inequality are multiplied by *B*. Changing the form of this constraint depends on the sign of *B*. Therefore, this inequality is considered in two domains: $D_B>0$ and $D_B<0$ and the relevant four forms are as follows:

1. $A-(cv)B>0$	if $D_B>0$,	2. $-A-(cv)B>0$	if $D_B>0$
3. $(cv)B-A>0$	if $D_B<0$,	4. $(cv)B+A>0$	if $D_B<0$

These forms can be easily solved by the method which will be explained in Section IV.B. In the constraint $A\div B>const$, if $B.b<0$ and $B.t>0$ then the constraint should be solved in the domain of $D_B = (0+\varepsilon, B.t)$. If it is not solved, then the constraint is solved in the domain of $D_B = (B.b, 0-\varepsilon)$, where the value of ε is considered close to zero to avoid the error of dividing by zero.

B. Satisfying the Complex Constraints

The main idea in solving complex statements is divide-and-conquer. To solve complex expression, first, the constraint is restated as one of five fundamental forms by means of variable substitution. Then, the initial domain of each substituted variable is obtained by respected expression. Afterwards, the simplified constraint is solved with one of the previously mentioned methods. Finally, with obtaining the new domain of each substituted variable, these domains will be added to the corresponding expressions of variables to form a new constraint. Therefore, in each step, the constraint is divided until it reaches to one of fundamental constraints. In order to clarify the proposed algorithm, five complex standard expressions will be examined.

1) *Satisfying* $\pm A\pm B>0$

This constraint can be considered separately as follows:

1. $A-B>0$,	3. $B-A>0$
2. $-A-B>0$,	4. $B+A>0$

The first constraint can be solved by the method which we proposed in the section 3). For other constraints, a set of replacement should be arranged along which *A* is replaced with *-A*\` in condition 2, *A* is exchanged by *B* in condition 3, and *B* is replaced with *-B*\` in condition 4. Finally, new domain for *A* and *B* are calculated based on the new domain of *A*\` and *B*\`.

2) *Satisfying* $\pm A\pm B>const$

This constraint can be solved in two different manners which are: (1) $\pm A+B>const$ and (2) $\pm A-B>const$. In the first form, we utilize $B\text{\`}=B-const$ and the initial domain for *B*\` is $D_B\text{\`} = (D_B.b-const, D_B.t-const)$. Domain of *A* is found after solving this resultant constraint. With respect to the new domain of *B*\`, new domain of B will be obtained as $D_B = (newD_{B\text{\`}}.b+const, newD_{B\text{\`}}.t+const)$. In order to solve the form (2), The variable substitution $B\text{\`}=B+const$ is required.

3) *Satisfying* $\pm c_1X_1\pm c_2X_2\pm\ldots\pm c_nX_n>const$

In order to solve this constraint, where c_i is a positive value, variable substitution is highly demanded. Therefore, the following variable replacements are utilized

$$A = \pm c_1X_1\pm c_2X_2\pm\ldots\pm c_{n-1}X_{n-1}, \qquad B = \pm c_nX_n.$$

Another necessity for satisfying this constraint is to specify initial domain of *A* and *B*. After solving $A+B>const$, new domains of *A* and *B* will add two new constraints to the corresponding expression. This procedure continues in an iterative manner until new constraints appear as one of fundamental constraints. Table 1 tabulates a pseudo code to find the initial domain of substituted variable.

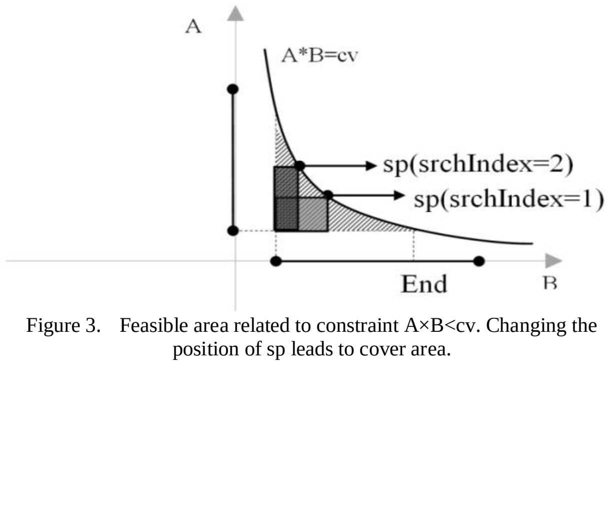


Figure 3. Feasible area related to constraint A×B<cv. Changing the position of sp leads to cover area.

TABLE I. CALCULATING THE INITIAL DOMAIN OF SUBSTITUTED VARIABLE.

```
domain obtainDomain(inputVariable){
    // inputVariable is in the form of $\pm c_1X_1 \pm c_2X_2 \pm \ldots \pm c_nX_n$
    // Dom is the new domain for the input variable
    // The domain of $X_i$ is define as $D_{Xi}$ = (bottom, top)

    Dom.top=0;          Dom.bottom=0;
foreach($X_i$) {
            if($X_i$.sign=='-')
            {
                Dom.top+=-$c_i \times D_{xi}$.bottom;
                Dom.bottom+=-$c_i \times D_{xi}$.top;}

            else
            {
                Dom.top+= $c_i \times D_{xi}$.top;
                Dom.bottom+= $c_i \times D_{xi}$.bottom; }
        }//end of foreach($X_i$)
        return  Dom;
}
```

After obtaining new domain for A and B, the constraint *A+B>const* is solved and in each iteration new domain for *A* (*newD*$_A$) and *B* (*newD*$_B$) are calculated. In fact, one of the following changes may happen for a variable like *A*:

1. The lower bound of variable domain is fixed and its upper bound decreases. In this situation a new condition should be solved that is $A<newD_A.top$ and is restated as $-A>-newD_A.top$.
   $newD_A.top<D_A.top,\ newD_A.bottom=D_A.bottom$
2. The upper bound of variable domain is fixed and its lower bound increases. In this situation a new condition should be solved that is $A>newD_A.bottom$.
   $newD_A.top=D_A.top,\ newD_A.bottom>D_A.bottom$
3. The variable domain is fixed and in this condition, there is no extra constraint related to the substitution variable.

It is worth pointing out in this context that the above conditions are also true for other variables like *B*. Moreover, in the proposed method, an expression with *n* terms is divided into two constraints where the first one has *n-1* terms and the second one has *1* term. This iterative method continues in order to simplify the constraint until it reaches the fundamental constraint.

4) Satisfying $\pm cx_1^{P1}(\times/\div)\ x_2^{P2}\ldots(\times/\div)\ x_n^{Pn} > const$

The expression $\pm cx_1^{P1}(\times/\div)\ x_2^{P2}\ldots..(\times/\div)x_n^{Pn}$ in fact, is a term of formula (1), in which *c* is a positive constant, x_i is a variable, and P_i is the power of *x*. In a situation in which the constraint becomes simplified, we have the following constraint:

$$\pm\ x_1^{P1}(\times/\div)\,x_2^{P2}\ldots..(\times/\div)\,x_n^{Pn}>cv$$

In order to satisfy this constraint, we propose a method where the substituting process is as follows:

$$A = \pm\ x_1^{P1}(\times/\div)x_2^{P2}\ldots(\times/\div)x_{n-1}^{Pn-1}, \qquad B = \pm x_n^{Pn} \tag{5}$$

As it is obvious, based on the equation (5), the constraint can be restated as fundamental constraint $\pm A\times B>cv$ and/or $\pm A\div B>cv$, which was discussed in section IV.A. as well.

For defining the initial domain of *A* and *B*, the pseudo code, which is tabulated in Table 2, is utilized.

TABLE II. CALCULATING DOMAIN OF SUBSTITUTION VARIABLES OF CONSTRAINT $\pm X_1(\times/\div)\ X_2(\times/\div)X_3\ldots..(\times/\div)X_N>CV$

```
domain obtainDomain(inputVariable)
{
//$D_{Xi}$ is the initial domain of $X_i$
    d.top= $D_{x1}$.top;
    d.bot = $D_{x1}$.bot;
    foreach  $x_i$    (i>1) {
        if ($P_i$>1) {
            $D_{xi}$.bot = min (power($D_{xi}$.bot, $p_i$), power($D_{xi}$.top, $p_i$));
            $D_{xi}$.top = max (power($D_{xi}$.bot, $p_i$), power($D_{xi}$.top, $p_i$)); }
        if ($op_{i-1}$=='×') {
            value[1] =d.top ×$D_{xi}$.bot;  value[2] =d.top ×$D_{xi}$.top;
            value[3] =d.bot× $D_{xi}$.bot; value[4] = d.bot×$D_{xi}$.top;
            d.top = max(value[]);    d.bot = min(value[]);
            }
        elseif ($op_{i-1}$=='÷') {
            if($D_{xi}$.bot==0)
                $D_{xi}$.bot=ε;   // to avoid the error of divide by zero
            elseif ($D_{xi}$.top==0)
                $D_{xi}$.top=- ε; // to avoid the error of divide by zero
            if ($D_{xi}$.bot<0 && $D_{xi}$.top>0) {
                value[1]=d.top÷ ε; value[2]=d.bot÷ ε;
                value[3]=d.top÷(- ε); value[4]=d.bot÷(- ε); }
            else {
                value[1]=d.top÷$D_{xi}$.bot; value[2]=d.top÷$D_{xi}$.top;
                value[3]=d.bot÷$D_{xi}$.bot; value[4]= d.bot÷$D_{xi}$.top; }
            d.top=max(value[]); d.bot=min(value[]);
        }
    }// end of foreach($X_i$)
    return  d;
}
```

After solving the expression, new domain of *A* and *B* will be obtained in one of the forms of statements which were discussed in the previous section. Finally, with respect to the new state of domain *A* and *B*, the corresponding constraint is specified and is solved.

5) Satisfying polynomial constraint

A polynomial constraint is similar to the constraint $\pm c_1X_1 \pm c_2X_2 \pm \ldots \pm c_nX_n > const$ where each variable is substituted with a term of variable(s). In order to satisfy polynomial constraint, we can utilize the methods which explained in satisfying $\pm c_1X_1 \pm c_2X_2 \pm \ldots \pm c_nX_n > const$. Then, each term of the constraint can be completely solved by method which was explained in satisfying $\pm cx_1^{P1}(\times/\div)\ x_2^{P2}\ldots..(\times/\div)\ x_n^{Pn} > const$.

V. EXPERIMENTAL RESULTS

In order to prove the efficiency of the proposed method, firstly, we calculate the time complexity; then, an experiment is conducted to demonstrate success of ADDR in covering feasible area in comparison with DDR.

Time complexity of an expression cannot be calculated unless the time complexity of each term is calculated. With respect to the proposed algorithm, the time complexity of a term is calculated as follows (6):

$$\begin{gathered} T(n) = (T(n-1)+T(1)) \times maxSI \\ T(1) = d \end{gathered} \tag{6}$$

where *T(n)* is the time complexity of a term with *n* variables, *maxSI* is the supreme value of *srchIndex*, and *d* is the time of solving $\pm A^p > const$. Therefore, the time complexity of solving one term is $O(maxSI^{\,n})$. With respect to the proposed recursive method in satisfying polynomial constraints, time complexity of solving a polynomial is calculated as follows (7):

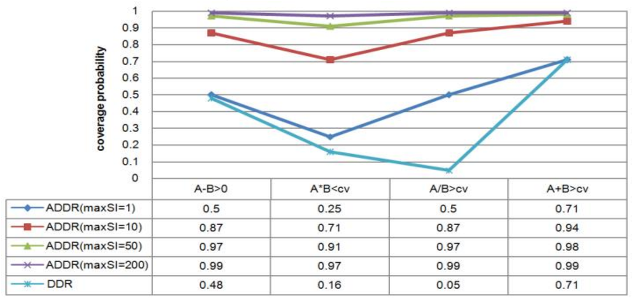
$$\begin{aligned} T(m) &= (T(m-1)+T(1)) \times maxSI \\ T(1) &= O(maxSI^{\,n}) \end{aligned} \tag{7}$$

where m is the number of terms. Therefore, the time complexity of ADDR is $O(maxSI^{\,m+n})$. With a view to this fact that the most polynomial constraints have a few terms and few variables the time complexity is reasonable.

The main purpose of ADDR is to cover all feasible area for the polynomial constraints utilizing split point. We compare ADDR and DDR in terms of satisfying polynomial constraints and calculate the probability of covering the feasible area in each method. Suppose C is a complex polynomial constraint which is composed of fundamental constraints such as $S_1, S_2, \ldots, S_n$. These fundamental constraints are those ones which are utilized in DDR to solve polynomial constraints. If P_{S_i} shows the probability of fundamental constraint coverage, then the probability of covering feasible area can be calculated by formula (8). Therefore, it is sufficient to calculate the probability of covering feasible area for the fundamental constraints.

$$P_C = \prod_{i=1}^{n} P_{S_i} \tag{8}$$

Fig. 4 shows the probability of feasible area coverage of two competitive methods, i.e., ADDR and DDR. As it is obviously seen, four fundamental constraints in ADDR with different *maxSI* covers the feasible area much better than DDR method. This experiment shows that ADDR has the capability of covering the feasible area utilizing split point and choosing adequate value for *maxSI*.



	A-B>0	A*B<cv	A/B>cv	A+B>cv
ADDR(maxSI=1)	0.5	0.25	0.5	0.71
ADDR(maxSI=10)	0.87	0.71	0.87	0.94
ADDR(maxSI=50)	0.97	0.91	0.97	0.98
ADDR(maxSI=200)	0.99	0.97	0.99	0.99
DDR	0.48	0.16	0.05	0.71

Figure 4. Probability of feasible area coverage for the fundamental constraint in DDR and ADDR. (In this experiment the initial domain for *A* and *B* is (1, 50) and *cv* =20.)

Despite the fact that choosing large value for maxSI can improve the coverage percent, in a case that the selected path from CFG is infeasible then the larger value just wastes the time and resources. In fact, if the tester program faces an infeasible path, it is unable to solve it even if the value of maxSI increases.

VI. Conclusion

After years of significant research into software testing, many application administrators would agree that the methods which are proposed for test data generation might work undesirably. In this work, the Constraint Based Testing (CBT) and Dynamic Domain Reduction (DDR) were examined, which embody the test data generation in satisfying polynomial and complex constraints. It is indicated that these methods have not enough capability of solving such constraints. Therefore, a heuristic for the improvement of DDR, which would be called Adaptive Dynamic Domain Reduction (ADDR), is introduced.

To achieve this intent for the generating test data, divide-and-conquer method is applied. ADDR is the solution to all obstacles related to CBT and DDR in solving polynomial constraint. In fact, the main contribution of ADDR is not only satisfying a polynomial constraint in a sensible time, but also being capable of fairly covering the feasible area through finding all solver vectors of the present and incoming conditions.

In order to demonstrate the efficiency of the proposed method, a set of experiments is conducted. These experiments show the proposed method can further improve the process of test data generation in a situation in which complicated or simultaneous constraints may result in the automation failure.

We plan to explore more challenges related to these issues such as solving mathematical function expressions and finding an objective function for solving *maxSI* in future works.

References

[1] G. Myers, “The art of software testing”, JohnWiley and Sons, second edition, 2004.

[2] I. Burnstein, “Practical software testing : a process-oriented approach”, Springer, 2003.

[3] A. Kumar, S. Tiwari, K. Mishra and A . Misra “Generation of efficient test data using path selection strategy with elitist GA in regression testing”. In: third IEEE International Conference on Computer Science and Information Technology, pp 389--393, 2010.

[4] S . Naghdali Zanjani, M. Dehghan Takht Fooladi and A. Bagheri “Improving Path Selection by Handling Loops in Automatic Test Data Generation”, IEEE 14th International Multitopic Conference, pp.273-278,2011.

[5] A.J. Offutt,Z. Jin and J. Pan “The dynamic domain reduction procedure for test data generation”, Software Practice And Experience, John Wiley and Sons,vol. 29,pp. 167-193, 1999.

[6] R. Demilli and J. Offutt “Constraint-based automatic test data generation”, IEEE Transactions on Software Engineering, vol. 17, pp. 900-910, 1991.

[7] Y. Jia and M. Harman, “An analysis and survey of the development of Mutation Testing”, IEEE Transactions on Software Engineering, vol. 2008, pp. 1-32 , 2010.

[8] M. Xiao, M. El-Attar, M. Reformat, and J. Miller, “Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques”, Empirical Software Engineering, Springer, vol. 12, pp.183-239 ,2007.

[9] R. Majumdara and R. Xu, “Directed Test Generation Using Symbolic Grammars”. Time, ACM ,2007.

[10] J. Berdine and C.Calcagno, “Symbolic Execution with Separation Logic”, In: APLAS, Springer, 2005.

[11] Korel, B.: “Automated software test data generation”, IEEE Transactions on Software Engineering, vol. 16, pp. 870–879 ,1990.

[12] N. Mansour and M. Salame, “Data Generation for Path Testing”, Journal of Software Quality, Springer, vol. 12, pp. 121-136, 2004.