International Journal of Database Theory and Application
Vol.7, No.1 (2014), pp.11-24
http://dx.doi.org/10.14257/ijdta.2014.7.1.02

# An Intelligent Method for Test Data Generation Based on Optimized Interval Arithmetic

Ying Xing[1,2], Yun-Zhan Gong[1], Ya-Wen Wang[1,3] and Xu-Zhou Zhang[1]

[1]*State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, 100876, China*
[2]*Liaoning Technical University, Huludao, 125105, China*
[3]*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China*

***Abstract***

*Path-oriented test data generation is in essence a Constraint Satisfaction Problem solved by search strategies, among which backtracking algorithms are widely used. In this paper, the backtracking algorithm Branch & Bound is introduced to generate path-oriented test data automatically. A model based on state space search is proposed to construct the search tree dynamically. Aiming at the programs containing constraints of strongly related variables even equalities, the static analysis technique interval arithmetic is optimized for the precise judgment of the assignment to each variable. The analysis on conflict is made accurate via distance for further domain reduction, thus ensuring the precise direction of the next search step. Experiments show that the proposed method outperformed other methods used in static test data generation. Specifically, it produces excellent results when variables are strongly related even when they are in equalities, and generation time increases stably and linearly with the increment of number of expressions including both equalities and inequalities.*

**Keywords:** *test data generation; Constraint Satisfaction Problem; Branch & Bound; state space search; interval arithmetic*

## 1. Introduction

Software testing plays an irreplaceable role in the process of software development, because it is an important stage to guarantee software reliability [1, 2]. And as the most important coverage testing [3], the automation of path-oriented test data generation is crucial in the testing process [4]. It can be solved by many methods [5-16], wherein the static method [5-9] is an important branch. Due to its utilization of static analysis techniques including symbolic execution [17, 18] and interval arithmetic [19, 20] without actually executing the program under test (PUT), the process of generating test data is definite with relatively less cost. It abstracts the constraints to be satisfied, and propagates and solves these constraints to obtain the test data. When the variables in the constraints are weakly related, test data can be generated rather well [21]. But when the variables are strongly related especially in equalities [22], the constraints become difficult to solve, which in turn poses problems in generating test data for PUTs containing this kind of constraints.

In this paper, we propose a new method for static test data generation based on our previous work, which tests programs written in C programming language with complex data structure.

Solving the constraints containing strongly related variables even equalities is our focus. The main contribution of this paper is as follows.

1) The problem of path-oriented test data generation is defined as a Constraint Satisfaction Problem(CSP), the solution space is represented as state space, and the backtracking algorithm Branch & Bound (BB) is introduced to solve the constraints.

2) Aiming at the difficulty in solving the constraints containing strongly related variables including equalities, interval arithmetic is enhanced to judge the assignment to each variable more precisely. The analysis on conflict is made accurate via distance for the further domain reduction in the next search step.

The rest of this paper is organized as follows. Section 2 provides the background underlying our research. Section 3 illustrates the proposed search algorithm in detail. Optimized interval arithmetic is described in Section 4 with a case study. Experimental analyses and empirical evaluations on the proposed algorithm are presented in Section 5. Section 6 concludes this paper and highlights directions for future research.

## 2. Background

Many forms of static test data generation make reference to the control flow graph (CFG) of the PUT [23]. In this paper, a CFG for a program $P$ is a directed graph $G=(N, E, i, o)$, where $N$ is a set of nodes, $E$ is a set of edges, and $i$ and $o$ are respective unique entry and exit nodes to the graph. Each node $n \in N$ is a statement in the program, with each edge $e=(n_r, n_t) \in E$ representing a transfer of control from node $n_r$ to node $n_t$. Nodes corresponding to decision statements such as *if* statements are branching nodes. Outgoing edges from these nodes are referred to as branches. A path through a CFG is a sequence $p=(n_1,n_2,\ldots,n_q)$, such that for all $r$, $1 \le r < q$, $(n_r,n_{r+1}) \in E$.

A path $p$ is regarded as feasible if there exists a program input for which $p$ is traversed, otherwise $p$ is regarded as infeasible. The path-oriented test data generation problem can be reformulated as a CSP [24]. $X$ is a set of variables $\{x_1, x_2,\ldots, x_n\}$, $D=\{D_1, D_2,\ldots, D_n\}$ is a set of domains[25,26] (a domain is composed of one or more intervals and an interval is a continuous range of values), and $D_i \in D$ ($i=1,2,\ldots,n$) is a finite set of possible values for $x_i$. For each path, $D$ is defined based on the variables' acceptable ranges. One solution to the problem is a set of values to instantiate each variable inside its domain denoted as $\{x_1 \mapsto V_1, x_2 \mapsto V_2, ..., x_n \mapsto V_n\}$, $V_i \in D_i$ to make path $p$ feasible. To be specific, each constraint defined by the PUT along $p$ should be satisfied. In static analysis, the feasibility of a path is judged by the result of interval arithmetic. To be more exact, the path is feasible only when all the constraints along the path are satisfied, which is the very reason why we optimize interval arithmetic to be more precise.

To better illustrate path-oriented test data generation, a simple example with a program *test1* and its corresponding CFG are shown in Figure 1, where *if_out_4*, *if_out_5*, and *exit_6* are dummy nodes. Adopting statement coverage, there is one path to be traversed, which is *Path 1:0→1→2→3→4→5→6* as shown in bold. The numbers along the paths denote nodes rather than edges of the CFG. To cover *Path 1*, we need to select $V=\{V_1,V_2\}$ from $\{D_1,D_2\}$ for $x_1$ and $x_2$, so that when executing *test1* using $\{V_1,V_2\}$ as an input, the path traversed is *Path1*. There are two branching nodes *if_head_1* and *if_head_2* along *Path1*, and two corresponding branches *T_1* and *T_2* containing the constraints to be satisfied.

```
   void test1(int x1, int x2)

1{ if(x1+x2==100)

2     if(x1-x2==20)

3          printf("Solved! ");

   }
```

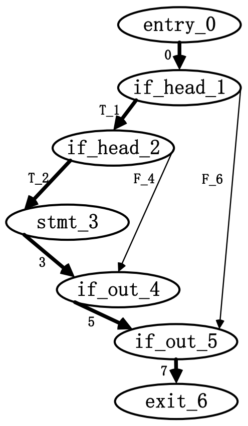

**Figure 1. Program *test1* and its corresponding CFG**

A CSP is generally solved by search strategies, among which backtracking algorithms [27] are widely used. In this paper, the process of exploring the solution space is represented by state space search [28] in artificial intelligence. This representation will facilitate the implementation of the search methods. With the aid of intelligent rules for selecting nodes to explore and pruning those that do not lead to a solution, the complexity of the search can be drastically reduced as compared to that of an exhaustive or enumerative search.

BB [29] is introduced to tackle the problem mentioned above, which is an efficient backtracking algorithm for searching the solution space of a problem. Considering that one solution is enough for path-oriented test data generation, Best-First-Search is our first choice. To find the 'Best', permutation of variables is required for branching to prune the branches stretching out from unneeded variables. Besides, as the domain of a variable is a finite set of possible values which may be quite large, bounding is necessary to cut the unneeded or infeasible solutions. Bounding is the focus of this paper and we propose Best-First-Search Branch and Bound (BFS-BB) to generate the test data automatically.

All the variables involved in BFS-BB are symbolic variables. During the search process, variables are divided into three sets: past variables (short for *PV*, already instantiated), current variable (now being instantiated), and future variables (short for *FV*, not yet instantiated). In addition, although the experiments were carried out on programs of different data types, integer variables are used as example in the following algorithms in order to simplify the explanations.

## 3. The Search Algorithm

### 3.1 State space search

The *state space* is a quadruple($S$, $A$, $I$, $F$), where $S$ is a set of states, $A$ is a set of arcs or connections between the states that correspond to the steps or operations of the search at different states, $I$ is a non-empty subset of $S$ denoting the initial state of the problem, and $F$ is a non-empty subset of $S$ denoting the final state of the problem.

A *state* is a tuple (*Precursor*, *Variable*, *Domain*, *Value*, *Type*, *Queue*). In a certain stage of the search process, for current state $S_{cur}$, *Precursor* provides a link to the previous state; *Variable*$=x_i \in X(i=1,2,\ldots,n)$ is the current variable; *Domain*$=D_{ij} \subseteq D_i \in D$, $(i=1,2,\ldots,n;$ $j=1,2,\ldots,m)$ in the form of [*min*, *max*] is the set of possible values that may be selected to instantiate *Variable*, where *min* and *max* are the lower and upper bounds respectively; *Value*=$V_{ij}\in D_{ij}$ is a value selected from *Domain*; *Type* marks the type of *state* which might be *active*, *extensive* or *inactive*; and *Queue* is a sequence of variables corresponding to $S_{cur}$.

*State space search* is all about finding one final state in a state space (which may be extremely large). 'Final' means that every variable has been instantiated with a definite value and the path to be traversed is proved to be feasible with all these values by interval arithmetic. At the start of the search *Precursor* is null, and when *Queue* is null the search ends. The path made up of all the *extensive* nodes in the search tree makes the solution path. The process of generating test data for path *p* takes the form of state space search. The state space needs to be searched to find a solution path from an initial state to a final state. We can decide where to go by considering the possible moves from the current state $S_{cur}$ and trying to look ahead.

## 3.2 Details of the proposed algorithm

The idea of our algorithm is to extend partial solutions. At each stage, a variable in *FV* is selected and assigned a value from its domain to extend the current partial solution. Interval arithmetic evaluates whether such an extension may lead to a possible solution of the CSP and prunes subtrees containing no solutions based on the current partial solution. The algorithm BFS-BB is expressed by pseudo-code as follows with two stages, which are initialization and state space search.

### Algorithm1. Best-first-search branch and bound

```
Input    p : the path to be traversed
Output result <Variable, Value>: test data making p feasible
Stage 1: Initialization
1: result <Variable, Value> = null;
2: $Q_1$←permutate FV;
3: $x_1$←head ($Q_1$);
4: select $V_{11}\in D_{11}$ for $x_1$;
5: initial state= (null, $x_1$, $D_{11}$, $V_{11}$, active,$Q_1$);
6: $S_{cur}$= initial state;
Stage 2: State space search
Begin
7: foreach $S_{cur}$ =(Pre, $x_i$, $D_{ij}$, $V_{ij}$, active, $Q_i$) do
8:successful=false;
9: call algorithm2. Optimized interval arithmetic;
10: if (successful= =false)
11:       if(|$D_{ij}$|= =1||j>=m)
12:             $S_{cur}$ = (Pre,$x_i$, $D_{ij}$, $V_{ij}$, inactive, $Q_i$);
13:             Pre = $S_{cur}$;
14:             $S_{cur}$ = (Pre, $x_i$, $D_{ij}$, $V_{ij}$, active, $Q_i$);
15:             remove<$x_i$,$V_{ij}$> from result;
16:             remove $x_i$ from PV;
17:       else j++;
18:             if(distance<0)
19:                   $D_{ij}$=[$V_{ij}$+1, $V_{ij}$+|distance|];
20:             else $D_{ij}$=[$V_{ij}$-|distance|,$V_{ij}$-1];
21:             select $V_{ij}\in D_{ij}$ for $x_i$;
22:             $S_{cur}$ = (Pre ,$x_i$, $D_{ij}$, $V_{ij}$, active, $Q_i$);
23: else $S_{cur}$ = (Pre ,$x_i$, $D_{ij}$, $V_{ij}$, extensive, $Q_i$);
24:         add < $x_i$, $V_{ij}$ > to result;
24:         update FV;
25:         $Q_i$←permutate FV;
26:         if($Q_i$!=null)
27:               $x_i$←head ($Q_i$);
28:               select $V_{i1}\in D_{i1}$ for $x_i$;
29:               Pre= $S_{cur}$;
30:               $S_{cur}$ =(Pre ,$x_i$,$D_{i1}$,$V_{i1}$,active, $Q_i$);
31:         else $S_{cur}$ =final state;
```

```
32: **endfor**
33: **return** *result*;
**End**
```

The first stage is to perform the initialization operations. At first, the table *result* storing test data is null. All the relevant variables in *FV* are permutated to form a queue $Q_1$ and its head $x_1$ is determined to be the best or the first variable to be instantiated. The initial value $V_{11}$ is selected from the domain $D_{11}$. With all these, the initial state is constructed as (*null*, $x_1$, $D_{11}$, $V_{11}$, *active*, $Q_1$), which is also the current state $S_{cur}$.

The second stage implements state space search. On constructing each state, *Type* is *active*, *Queue*=$Q_i$, and *Variable*= $x_i$ is the head of $Q_i$. To each current state (*Pre*, $x_i$, $D_{ij}$, $V_{ij}$, *active*, $Q_i$), optimized interval arithmetic is carried out to determine the direction of the next search step. If it succeeds, *Type* becomes *extensive*, variables in *FV* will be permutated to get *Queue*= $Q_i$, $S_{cur}$ becomes *Precursor*, and the head of $Q_i$ will be *Variable* of next state. With all these, a new state can be constructed for which to continue interval arithmetic. If after the conduction of interval arithmetic, no variable needs to be permutated, then all the variables have been assigned the right values to make *p* feasible. *result* is returned as the test data .

If the interval arithmetic for a certain state meets a conflict, then *Type* remains *active*, analysis on the conflict information is utilized to reduce $D_{ij}$, where *Value* is reselected for $x_i$. In that case, the search will expand to a state with a different value for the same variable. If all values within its domain for the same variable are tried out or the number of interval arithmetic has reached the upper bound *m* (the threshold used to control the breadth of the search tree), then$<x_i,V_{ij}>$ is removed from *result* and $x_i$ from *PV*, and *Type* becomes *inactive*, so the search will have to backtrack to *Precursor* at the higher level of the search tree.

## 4. Optimized Interval Arithmetic

### 4.1 The design of the algorithm

It has been previously studied that interval arithmetic works well on a single path [21]. On this base, we optimize interval arithmetic by dividing the path to be traversed into its basic constituents, in our case, the constraints, or the branching conditions (see Definition 3)，which are then considered in the sequence according to their ordering on the path. The domain of the current variable $x_i$ ($D_{ij}$) is involved in interval arithmetic as part of the domain of all variables (*D*). The optimization is primarily used to solve the strong constraints between variables especially equalities. They are stricter than inequalities in test data generation，because the values assigned to variables affect each other when trying to satisfy the constraints defined by equalities. Besides, a library of inverse functions is added in case of the occurrences of library functions in the PUT.

**Definition 3.** Let *B* be the set of boolean values{*true*, *false*}, *D* be the domain of all variables( $V_{ij} \in D_{ij} \subseteq D_i \in D$ ), the ***branching condition*** *Br($n_q$,$n_{q+1}$): D→B* where $n_q$ is a branching node is defined as the following formula:

$$Br(n_q, n_{q+1}) = \begin{cases} true\text{, if } (n_q, n_{q+1}) \text{ is traversed with } D \\ false\text{, otherwise} \end{cases} \qquad (1)$$

Hence, for the *k* branching nodes along the path, all *k* branching conditions should be *true* to make the path feasible if *p* is traversed with *D*. But if less than *k* branching conditions are satisfied, then the branch with *false* branching condition should be spotted and the conflict information should be analyzed, so as to reduce the domain of current variable $x_i(D_{ij})$. The value of the branching condition $Br(n_{qa},n_{qa+1})(a\in[1,k])$ depends on two factors: 1) $D^a$ ,which is the domain of all variables that satisfies all the $a$-1 branching conditions ahead and will be used as input for the calculation of the $a$th branching condition; 2) $\tilde{D}^a$ , which is the result when calculating $Br(n_{qa},n_{qa+1})$ with $D^a$ and satisfies the $a$th branching condition. $D^a \cap \tilde{D}^a \neq \varnothing$ means that $D^a \cap \tilde{D}^a$ satisfies both all the $a$-1 branching conditions ahead and the $a$th branching condition, ensuring that interval arithmetic can continue to calculate the remaining branching conditions.

The calculating process of optimized interval arithmetic is shown in Figure 2. There is no conflict in (a), and all the $k$ branching conditions are satisfied, so the permutation of the remaining variables follows in the search algorithm. On the contrary, there is a conflict in (b), and the $h$th ($1\le h\le k$) branching condition is not satisfied, so the analysis on the conflict follows in the search algorithm. The calculating process of optimized interval arithmetic is shown by pseudo-code as follows.

## Algorithm 2. Optimized interval arithmetic

```
Input   $D^1$:the domain of all variables; $Br(n_{qa},n_{qa+1})(a\in[1,k])$: branching conditions along the path
Output $D^{k+1}$: the reduced domain of all variables; distance: the value used for the reduction of $D_{ij}$ after a conflict
Begin
1: for $a\to 1:k$
2:       $Br(n_{qa},n_{qa+1})=false$;
3:       $\tilde{D}^a \leftarrow$calculate $Br(n_{qa},n_{qa+1})$ with $D^a$;
4:       if ( $D^a \cap \tilde{D}^a \neq \varnothing$ )
5:            $Br(n_{qa},n_{qa+1})=true$;
6:            $D^{a+1} = D^a \cap \tilde{D}^a$ ;
7:       else $distance = V_{ij} - V_{ij}^{'} (V_{ij}^{'} \in \tilde{D}^a)$ ;
8:            return distance;
9: endfor
10:successful=true;
11:return $D^{k+1}$;
End
```

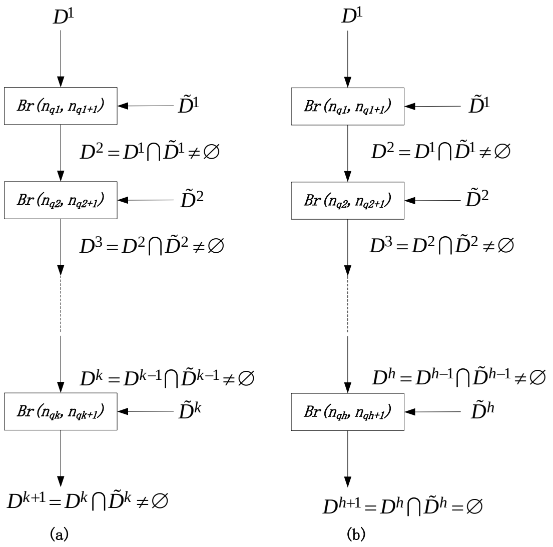
$D^1$
$Br(n_{q1}, n_{q1+1})$
$\tilde{D}^1$
$D^2 = D^1 \cap \tilde{D}^1 \neq \varnothing$
$Br(n_{q2}, n_{q2+1})$
$\tilde{D}^2$
$D^3 = D^2 \cap \tilde{D}^2 \neq \varnothing$
$D^k = D^{k-1} \cap \tilde{D}^{k-1} \neq \varnothing$
$Br(n_{qk}, n_{qk+1})$
$\tilde{D}^k$
$D^{k+1} = D^k \cap \tilde{D}^k \neq \varnothing$
(a)
$D^h = D^{h-1} \cap \tilde{D}^{h-1} \neq \varnothing$
$Br(n_{qh}, n_{qh+1})$
$\tilde{D}^h$
$D^{h+1} = D^h \cap \tilde{D}^h = \varnothing$
(b)

**Figure 2. The calculating process of optimized interval arithmetic (a) no conflict (b) conflict detected**

Interval arithmetic receives $V_{ij}$, the value of the current variable, which is part of the domain of all variables $D^1$ and evaluates the branching condition corresponding to the branch $(n_{q1},n_{q1+1})$ where $n_{q1}$ is the first branching node. The branching condition $Br(n_{q1},n_{q1+1})$ is generally not satisfied for all the values in $D^1$ but for values in a certain subset $D^2 \subseteq D^1$ ensuring the traversal of branch$(n_{q1},n_{q1+1})$, *i.e.*, $D^1 \xrightarrow{Br(n_{q1},n_{q1+1})} D^2$. Next the branching condition $Br(n_{q2},n_{q2+1})$ is evaluated given that the domain of all variables is $D^2$. Again, generally $Br(n_{q2},n_{q2+1})$is only satisfied by a subset $D^3 \subseteq D^2$, *i.e.*, $D^2 \xrightarrow{Br(n_{q2},n_{q2+1})} D^3$. This procedure continues along *p* until all the *k* branching conditions are satisfied with the domains of all variables reduced. The propagation of the constraints made up of the branching conditions along *p* takes the form of $D^1 \xrightarrow{Br(n_{q1},n_{q1+1})} D^2 \xrightarrow{Br(n_{q2},n_{q2+1})} D^3 ... D^k \xrightarrow{Br(n_{qk},n_{qk+1})} D^{k+1}$, where $D^1 \supseteq D^2 \supseteq D^3 ... \supseteq D^k \supseteq D^{k+1}$.

If in this process $Br(n_{qh},n_{qh+1})$= *false* ($1 \le h \le k$), which means a conflict is detected, then interval arithmetic is aborted and reduction of $D_{ij}$ is carried out according to the result of interval arithmetic at the branch with the conflict. So we give the following definition.

**Definition 4.** Let $V_{ij}^{'}$ be the value of the current variable $x_i$ determined by interval arithmetic to make a branching condition to be *true* after a conflict occurs, ***distance*** is calculated by the following formula:

$$distance = V_{ij} - V_{ij}^{'} \qquad (2)$$

*Distance* lays the foundation for the reduction of $D_{ij}$, for it provides both the upper and the lower bounds of $D_{ij}$ after reduction，determined by the sign and absolute value of *distance* respectively. Since the reduction of $D_{ij}$ happens in two opposite directions, the efficiency of the algorithm is improved greatly. To $S_{cur}$=(*Pre*, $x_i$, $D_{ij}$, $V_{ij}$, *active*, $Q_i$), if a conflict is detected by interval arithmetic, then $D_{ij}$ is reduced where the next value to be assigned to $x_i$ is selected. Hence the constraints made up the branching conditions are propagated in a more and more precise manner.

## 4.2 Case study

The program *test* in Figure 1 is used as a case to illustrate the process of optimized interval arithmetic. In *test1*, there are two input variables *x1* and *x2*. The path to be traversed is *Path 1:0→1→2→3→4→5→6* that passes the two true branches of the *if* statements and reaches *stmt3*. The true branches *T_1* and *T_2* make the branching conditions and their corresponding branch predicates exactly the same，causing the process of test data generation to be in fact solving an equality set. To put it more exactly, only $\{x1 \mapsto 60, x2 \mapsto 40\}$ satisfies the two constraints, which is very strict for a PUT with two variables. The initial domains of *x1*and *x2* are both set [0,100].

Let $S_{cur}$=(*null*, *x1*, [0,100], 50, *active*, (*x1*,*x2*)), which is also the initial state. That is, *x1* is selected to be the first variable to be instantiated，and it is assigned 50 from [0,100]. Optimized interval arithmetic evaluates whether $D^1$= {*x1*:[50,50]; *x2*:[0,100]} satisfies the constraints along the path as shown in Figure 3. It can be seen that after the first constraint x1+x2=100 is satisfied, the domains of both variables are reduced to $D^2$={*x1*:[50,50]; *x2*:[50,50]}. But when evaluating the second constraint x1-x2=20 with $D^2$, a conflict is detected for $D^3$={*x1*:[50,50]∩[70,70]=Ø;*x2*:[50,50]∩[30,30]=Ø}, where the domain of the current variable *x1* is empty.

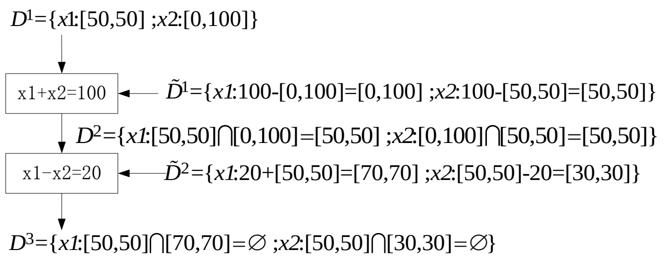
$D^1$={$x1$:[50,50] ;$x2$:[0,100]}
x1+x2=100
$\tilde{D}^1$={$x1$:100-[0,100]=[0,100] ;$x2$:100-[50,50]=[50,50]}
$D^2$={$x1$:[50,50]⋂[0,100]=[50,50] ;$x2$:[0,100]⋂[50,50]=[50,50]}
x1−x2=20
$\tilde{D}^2$={$x1$:20+[50,50]=[70,70] ;$x2$:[50,50]-20=[30,30]}
$D^3$={$x1$:[50,50]⋂[70,70]=∅ ;$x2$:[50,50]⋂[30,30]=∅}

**Figure 3. The calculating process of interval arithmetic when *x1* is assigned 50**

Interval arithmetic calculates *distance* of the current variable *x1* on the branch *T_2* with the conflict to be *-20* according to formula (2), that is, 50-70=-20. With *distance*<0 it can be concluded that 50 for *x1* is smaller for the satisfaction of the constraint on *T_2*, so the next value to be assigned to *x1* should be selected from [51,100] where the lower bound is determined to be 51. However, not every value in [51,100] will be better for the satisfaction of the constraint on *T_2* than 50. For example, 80 for *x1* is too large and is farther from the right value than 50. Therefore, it is necessary to determine a more suitable upper bound according to the absolute value of *distance*, which is 20. And finally the domain of *x1* is reduced to [51,70]. Then 60 from [51,70] is assigned to *x1*, and the current state $S_{cur}$ changes into (*null*, *x1*, [51,70], 60, *active*, (*x1*,*x2*)). Interval arithmetic evaluates whether $D^1$={*x1*:[60,60]; *x2*:[0,100]} satisfies the constraints along the path as shown in Figure 4.

It can be seen that after the first constraint x1+x2=100 is satisfied, the domains of both variables are reduced to $D^2$={*x1*:[60,60]; *x2*:[40,40]}. $D^2$ also satisfies the second branching condition x1-x2=20, and it is calculated that $D^3$ to be {*x1*:[60,60]; *x2*:[40,40]}. Since both constraints are satisfied with 60 for *x1*, *Type* is changed into *extensive*, and BFS-BB will take the next search step.

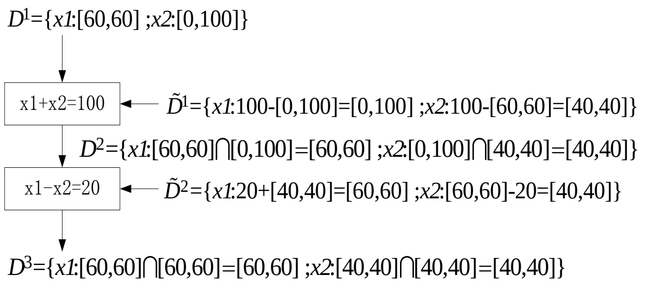
$D^1$={$x1$:[60,60] ;$x2$:[0,100]}
x1+x2=100
$\tilde{D}^1$={$x1$:100-[0,100]=[0,100] ;$x2$:100-[60,60]=[40,40]}
$D^2$={$x1$:[60,60]⋂[0,100]=[60,60] ;$x2$:[0,100]⋂[40,40]=[40,40]}
x1−x2=20
$\tilde{D}^2$={$x1$:20+[40,40]=[60,60] ;$x2$:[60,60]-20=[40,40]}
$D^3$={$x1$:[60,60]⋂[60,60]=[60,60] ;$x2$:[40,40]⋂[40,40]=[40,40]}

**Figure 4. The calculating process of interval arithmetic when *x1* is assigned 60**

## 5. Experimental Analyses and Empirical Evaluations

We carried out a large number of experiments in our team, where the PUT is automatically analyzed, its basic information is abstracted to form the Abstract Syntax Tree (AST), and its CFG is generated. According to the specified coverage criteria, the paths to be covered are generated and provided for BFS-BB as input. After test data have been generated by BFS-BB, the test drive is generated to provide the environment to execute the test case. There are some auxiliary functions in our team, including coverage observation, display of the covered code lines as well as the execution results, and the management of test cases for the convenience of regression testing.

The experiments were performed in the environment of MS Windows 7 with 32-bits, Pentium 4 with 2.8 GHz and 2 GB memory. The algorithms were implemented in Java and run on the platform of eclipse. The experiments include three parts. Section 5.1 tests the performance of BFS-BB using a test bed containing a few commonly used benchmarks. Section 5.2 presents a performance evaluation of optimized interval arithmetic. Section 5.3 compares BFS-BB with other static methods used in test data generation. The details of the programs used in the experiments are shown in Table 1.

**Table 1. Programs used in the experiments**

| Program | LOC | # of branches | # of variables | Description | Source |
|---|---|---|---|---|---|
| bonus | 29 | 10 | 1 | to calculate bonus according to profit | referring to[21] |
| days | 33 | 17 | 3 | to calculate which day a specific day is in a year | referring to[21] |
| statistics | 21 | 8 | 5 | to count the number of each kind of characters | referring to[21] |
| isValidDate | 59 | 16 | 3 | to check whether a date is valid or not | referring to[30] |
| gcd | 38 | 5 | 2 | to calculate greatest common denominator | referring to[31] |

### 5.1 Testing a composed test bed

In this part, test data were automatically generated to meet three control flow coverage criteria, which were statement, branch, and MC/DC. The test bed was a composed program with 402 lines, 29 input variables, and complex structure that might appear in real-world programs. Some commonly used benchmarks for test data generation such as *bonus*, *days*, *statistics*, and *isValidDate* were all used to compose this test bed.

The result is shown in Table 2. The numbers of paths were different owing to different coverage criteria adopted. BFS-BB was able to generate test data for all the feasible paths no matter which coverage criterion was taken. The MC/DC coverage did not reach 100%, because it is relatively strict and difficult to meet, and subsumes statement and branch coverage [32]. But tolerable coverage was achieved within tolerable time. There exists a trade-off between efficiency and success rate.

**Table 2. The coverage achieved by BFS-BB using a composed test bed**

| Adequacy criterion | # of paths | Average Coverage % |
|---|---|---|
| statement | 61 | 100 |
| branch | 119 | 100 |
| MC/DC | 125 | 98 |

### 5.2 Testing the performance of optimized interval arithmetic

Optimized interval arithmetic is primarily used for PUTs containing constraints of strongly related variables especially equalities. So in this part, experiments were carried out to evaluate the performance of BFS-BB for varying numbers of expressions including both inequalities and equalities. To be specific, our major concern is the relationship between generation time and the number of expressions. This was accomplished by repeatedly running BFS-BB on generated test programs having 50 input variables. Adopting statement coverage, in each test the program contained $u$ ($u \in [1,50]$) *if* statements (equivalent to $u$ path constraints or $u$ branching conditions) and there was only one path with entirely true branches (T,TT,TTT,…) to be traversed, i.e., all the branching conditions were the same as the corresponding predicates. The predicate of each *if* statement was an expression in the form of

$$[a_1, a_2, ..., a_{50}][x_1, x_2, ..., x_{50}]' \ rel\_op \ \ const[u] \qquad (3)$$

where $a_1, a_2, \ldots, a_{50}$ were randomly generated numbers either positive or negative, $rel\_op \in \{>, \geq, <, \leq, =, \neq\}$, and $const[u]$ ($u \in [1,50]$) was an array of randomly generated constants within [0,1000]. The randomly generated $a_v$($v$=1,2,…,50) and $const[u]$ should be selected to make the path feasible. This arrangement constructs the strongest linear relation between variables, all of which are relevant to the path to be traversed. The programs for various values of $u$ ranging from 1 to 50 were each tested 50 times and the time required to generate the data for each test was recorded. The results can be seen in Figure 5. For the sake of easy observation, the axes of generation time for both cases are normalized.

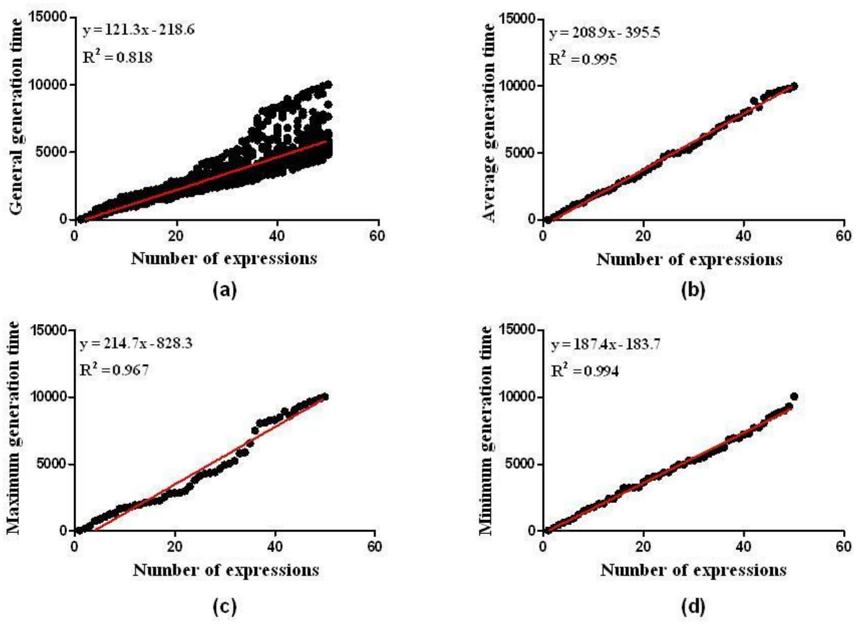

**Figure 5. Relationship between generation time and the number of expressions (*u*) for variables that are the strongest linearly related**

Figure 5 shows the relationship between generation time and the number of expressions ($u$) for variables that are the strongest linearly related. (a), (b), (c) and (d) represent four different situations marked by the ordinates. It can be seen that the generation time increases approximately linearly with the number of expressions and the linear correlation relationship is significant at 95% confidence level with p-value far less than 0.05. As the increase of the number of expressions, generation time increases at an even speed. The average value and the minimum value can be represented as straight lines very well, showing that they are the most ideal and stable in the four situations with the values of $R^2$ to be 0.995 and 0.994 separately.

The slopes of the fitting lines reflect the change of generation time. It can be seen that the slope of the fitting line in (d) corresponding to the minimum value of generation time is smaller than that in (b) corresponding to the average one, which, however, is smaller than that in (c) corresponding to the maximum one, all of which fit the experimental results. Variations between tests with the same values of $u$ were attributed to the randomness involved in 1) the difference in the selection of the initial values for the sake of diversity of test data; 2) the difference of the expressions along the path (it will surely take longer time to satisfy an equality than an inequality for equalities are stricter constraints).

### 5.3 Comparison with other static methods

This part presents the result from an empirical comparison of BFS-BB with the static method in [21], which performs well on programs containing variables that are weakly related. The comparison adopted three control flow criteria: statement, branch, and MC/DC.

The comparison result is shown in Table 3. It can be seen that BFS-BB reached 100% coverage for all the test beds using three coverage criteria while method in [21] did not. That is largely attributed to the intelligent methods utilized in BFS-BB, especially optimized interval arithmetic. There are modulus operations in the program *days*, so it had been difficult for the method in [21] to handle, causing loss of precision. But due to the library of inverse functions, it is not so difficult for BFS-BB.

**Table 3. Comparison result with method in [21] using three coverage criteria**

| program | Coverage criterion | # of paths | Average Coverage by method in [21] % | Average Coverage by BFS-BB % |
|---|---|---|---|---|
| bonus | statement | 6 | 25 | 100 |
| | branch | 6 | 37 | 100 |
| | MC/DC | 6 | 30 | 100 |
| days | statement | 17 | 100 | 100 |
| | branch | 17 | 100 | 100 |
| | MC/DC | 14 | 94 | 100 |
| statistics | statement | 4 | 100 | 100 |
| | branch | 5 | 100 | 100 |
| | MC/DC | 11 | 82 | 100 |
| gcd | statement | 3 | 85 | 100 |
| | branch | 3 | 77 | 100 |
| | MC/DC | 5 | 75 | 100 |

## 6. Conclusion

In the framework of state space search, this paper reformulates the problem of path-oriented test data generation as a CSP and introduces BB from artificial intelligence to solve this problem. Interval arithmetic is optimized to be more precise in the solving process. The conflict is analyzed precisely to distance for the further domain reduction in the next search step. Experimental results show that BFS-BB performs well for programs containing constraints of both strongly related variables and weakly related variables. It works especially well for programs containing equalities.

Our future research concerns not only how to generate test data to reach high coverage but how coverage criteria, generation approach, and system structure jointly influence test effectiveness. The MC/DC coverage criterion will be given more emphasis. The effectiveness of the generation approach continues to be our primary work.

## References

[1] R. Zhao and M. R. Lyu, "Character string predicate based automatic software test data generation", in Proceedings of IEEE the 3rd International Conference on Quality Software, ser. QSIC'03, Washington DC: IEEE Computer Society Press, **(2003)**, pp. 255-266.
[2] H. -Q. Zhao and S. Jing, "An algebraic model of service oriented trustworthy software architecture", Chinese Journal of Computers, vol. 33, no. 5, **(2010)** May, pp. 890-899.
[3] S. Raul and M. J. Harrold, "Demand-driven propagation-based strategies for testing changes", Softw. Test. Verif. Reliab., **(2013)**, pp. 499-528.
[4] J. Shan, J. Wang and Z. Qi, "Survey on path-wise automatic generation of test data", Acta Electronica Sinica, vol. 32, no. 1, **(2004)**, pp. 109-113 (in Chinese).
[5] A. Gotlieb, B. Botella and M. Rueher, "Automatic test data generation using constraint solving techniques", ACM SIGSOFT Software Engineering Notes, vol. 23, no. 2, **(1998)** March, pp. 53-62.
[6] Z. X. Xu and J. Zhang, "A test data generation tool for unit testing of C programs", in Proceedings of the 6th International Conference on Quality Software, ser. QSIC'06, Washington DC: IEEE Computer Society Press, **(2006)**, pp. 107-116.
[7] A. Gotlieb, "Euclide: a constraint-based testing framework for critical c programs", in Proceedings of the 2nd International Conference on Software Testing Verification and Validation, ser. ICST'09, Washington DC: IEEE Computer Society Press, **(2009)**, pp. 151-160.
[8] I. Chung and J. M. Bieman, "Generating input data structures for automated program testing", Softw. Test. Verif. Reliab., vol. 19, no. 1, **(1990)** March, pp. 3-36.
[9] J. Zhang, "Symbolic execution of program paths involving pointer and structure variables", in Proceedings of IEEE the 4th International Conference on Quality Software, ser. QSIC'04, Washington DC: IEEE Computer Society Press, **(2004)**, pp. 87-92.
[10] B. Korel, "Automated software test data generation", IEEE Trans. Softw. Eng., vol. 16, no. 8, **(1990)** August, pp. 870-879.
[11] P. Godefroid, "Compositional dynamic test generation", ACM SIGPLAN Notices, vol. 42, no. 1, **(2007)** January, pp. 47-54.
[12] X. Xie, B. Xu, S. Liang, *et al.*, "Genetic test case generation for path-oriented testing", Journal of Software, vol. 20, no. 12, **(2009)** December, pp. 3117-3136, (in Chinese).
[13] Y. Xue, C. Wei, W. Yongji, *et al.*, "An automated approach for structural test data based on messy GA", Journal of Software, vol. 17, no. 8, **(2006)** August, pp. 1688-1697, (in Chinese).
[14] W. Lin, Y. Feng and Z. Ruilian, "Path-oriented test data generation based on improved genetic algorithm", Computer Engineering, vol. 38, no. 4, **(2012)** February, pp. 158-161, (in Chinese).
[15] Z. Ruilian, "Search-based automatic path test generation method for character string data", Journal of Computer-Aided Design & Computer Graphics, vol. 20, no. 5, **(2008)** May, pp. 671-677, (in Chinese).
[16] K. Sen, D. Marinov and G. Agha, "CUTE: a concolic unit testing engine for C", in Proceedings of the 10th European Software Engineering Conference, ser. ESEC'05, New York: ACM Press, **(2005)**, pp. 263-27.
[17] J. C. King, "Symbolic execution and program testing", Communications of the ACM, vol. 19, no. 7, **(1976)** July, pp. 385-394.
[18] S. Person, G. Yang, N. Rungta, *et al.*, "Directed incremental symbolic execution", ACM SIGPLAN Notices, vol. 47, no. 6, **(2012)**, pp. 504-515.
[19] T. Hickey, Q. Ju and M. H. Van Emden, "Interval arithmetic: From principles to implementation", Journal of the ACM, vol. 47, no. 2, **(2001)**, pp. 1038-1068.
[20] W. Zhiyan and L. Chunyan, "The application of interval computation in software testing", Journal of Software, vol. 9, no. 6, **(1998)** June, pp. 438-443, (in Chinese).
[21] W. Yawen, G. Yunzhan and X. Qing, "A method of test case generation based on necessary interval set", Journal of Computer-Aided Design & Computer Graphics, vol. 25, no. 4, **(2008)** April, pp. 550-556, (in Chinese).

[22] M. J. Gallagher and V. L. Narasimhan, "Adtest: a test data generation suite for ada software systems", IEEE Trans. Softw. Eng., vol. 23, no. 8, **(1997)** August, pp. 473-484.
[23] P. McMinn, "Search-based software test data generation: a survey", Softw. Test. Verif. Reliab., vol. 14, no. 2, **(2004)** June, pp. 105-156.
[24] C. Xinguang and P. Van Beek, "Conflict-directed backjumping revisited", Journal of Artificial Intelligence Research, vol. 14, **(2001)** June, pp. 53-81.
[25] W. Yawen, G. Yunzhan, X. Qing and Y. Zhaohong, "Variable range analysis on interval computation", Journal of Beijing University of Posts and Telecommunications, vol. 32, no. 3, **(2008)** April, pp. 550-556, (in Chinese).
[26] W. Yawen, G. Yunzhan, X. Qing, *et al.*, "A method of variable range analysis based on abstract interpretation and its applications", Acta Electronica Sinica, vol. 39, no. 2, **(2011)** February, pp. 293-303, (in Chinese).
[27] E. G. Lisgara, G. I. Karolidis and G. S. Androulakis, "Advancing the backtrack optimization technique to obtain forecasts of potential crisis periods", Applied Mathematics, vol. 3, no. 30, **(2012)**, pp. 1538-1551.
[28] D. Szer, F. Charpillet and S. Zilberstein, "MAA: a heuristic search algorithm for solving decentralized POMDPs", In Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence, ser. UAI'05, Edinburgh, Scotland, **(2005)**, pp. 576–583.
[29] L. Gao, S. K. Mishra and J. Shi, "An extension of branch-and-bound algorithm for solving sum-of-nonlinear-ratios problem", Optimization Letters, vol. 6, no. 2, **(2012)**, pp. 221-230.
[30] C. Mao, Y. Xinxin and C. Jifu, "Generating test data for structural testing based on ant colony optimization", in Proceedings of IEEE the 12th International Conference on Quality Software, ser. QSIC'12, Washington DC: IEEE Computer Society Press, **(2012)**, pp. 98-101.
[31] Bouchachia and Abdelhamid, "An immune genetic algorithm for software test data generation", in Proceedings of the 7th International Conference on Hybrid Intelligent Systems. Washington DC: IEEE Computer Society Press, **(2007)**, pp. 84-89.
[32] A. Rajan, M. W. Whalen and M. P. E. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage", in Proceedings of the 30th ACM/IEEE International Conference on Software Engineering, ser. ICSE'08. New York, NY, USA: ACM Press, **(2008)**, pp. 161-170.