An efficient method of logical expression constraint solving in path-oriented unit test

Yalan Wang¹, Songyi Liu², Yawen Wang¹,* and Yunzhan Gong¹

¹State Key Laboratory of Networking and Switching Technology, Beijing university of Posts and Telecommunications, 100876, Beijing, China
²China Launch and Tracking Control, 100020, Beijing, China

*Email: wangyawen@bupt.edu.cn

Abstract. This paper extends the theory of constraint solving and presents a method to solve logical expression constraint problems existing in the process of generating test cases automatically. It defines the concept of the logical expression constraint on the path of C programs. In order to improve solving efficiency, it also presents an algorithm of pre-processing and heuristic backtracking. Our experiment results show that the method can narrow the domain of variables from meaningless infinite intervals to a finite range and achieve better coverage, and the results also show that the algorithm can improve the testing efficiency and halve the time of generating test cases.

1. Introduction
Path-oriented automatic generation of test cases is one of the typical ways in unit test, which depends on the establishment and solution of the constraint system. And the constraint system is mainly solved by the search method. The researchers have also done a lot of research on this method [1-3], from which it can be found that the problem of solving the logical expression constraint is less concerned. However, in a practical project, the logical expression constraints show relatively high proportion of the constrains (statistics from three large projects [4] - aa200c, qlib, and deco, which indicate that the number of logical expression constraints exceeds 30% of the total). Based on the existing interval algorithm [5] in CTS (Code Testing System), a method of logical expression constraints solving in C-language programs is proposed in this paper, which can support logical expression constraints solving and improve the efficiency.

2. Path-oriented logical expression constraint solving algorithm

2.1. Problem definition
The Constraint Satisfaction Problem (CSP) consists of a set of variables \( x_1, x_2, \ldots, x_n \), the domain \( R_1, R_2, \ldots, R_n \) corresponding to the variable and a set of constraints. Solving the CSP problem is to find the value of the variables that in the variable domain so that the constraint is satisfied. [7] The logical expression constraint satisfaction problem (LECSP) is a special constraint satisfaction problem for C language. So the input of LECSP consists of three elements: variable set \( V, V = \{v_1, v_2, \ldots, v_n\} \); the domain of the variable \( D, D = \{d_1, d_2, \ldots, d_n\} \); and the constraint between the variables \( LC, LC = \{Exp_1 rop Exp_2 rop \ldots rop Exp_n, Lv\}, rop \in \{\&\&|!\} \), \( Lv \in \{T,F\} \). \( Exp_i \) is a constraint composed of
mathematical expressions that connected by relational operators (=, <, >, <=, >=). We can also call it atomic expression constraint (AC). \( L_v \) is a logical value, when \( L_v \) is T, the true value of the constraint \( LC \) is True, and the logical value F represents False. The particularity of the logical expression constraint solving problem studied in this paper is not finding the value of variables, but making domain \( D \) reduction [6], so that when the variable \( V \) is searched in the domain \( D \), it can reduce the search space in the constraint solving process, for the reason that can improve the efficiency of generating testing cases.

2.2. Introduction of algorithm

In this paper, a framework is established for logical expression constraints solving, and three algorithms are proposed in this framework. The whole algorithm is shown in Algorithm 1. The algorithm solves the non-LCS constraints on the path \( p \) first and takes the result as the initial variable domain. Secondly, we will call the "pruning" algorithm and the Variable Level Determination Algorithm for pre-processing, and heuristic backtracking algorithm will be called if there is a conflict in the solving process.

Algorithm 1 Path-oriented logical expression constraint solving algorithm

```
Input: The path to be covered, \( p \).
Output: The variable domain that satisfies the logical expression constraint on path \( p \).
variableDomain:
1: procedure MPROCEDURE
2: \( CS \leftarrow \) All non-LCS constraint sets on path \( p \)
3: \( LCS \leftarrow \) All LCS constraint sets on path \( p \)
4: result \( \leftarrow \) NULL
5: initDomain \( \leftarrow \) intervalCompute(CS)
6: \( \text{for each} \ \text{lc} \in LCS \text{do} \)
7: \( \text{AC} \leftarrow \text{Get all atomic expression constraint sets that satisfy lc} \)
8: \( \text{AC} \leftarrow \text{getSimplification(AC)} \) //call "Pruning" algorithm
9: \( \text{AC} \leftarrow \text{getOrder(AC)} \) //call variable level determination algorithm
10: \( \text{AC} \leftarrow \text{getFirstSet(AC)} \)
11: \( \text{conflict} \leftarrow \text{true} \)
12: \( \text{while conflict false do} \)
13: \( \text{con/flag} \leftarrow \text{intervalCompute(AC)} \)
14: \( \text{if con/flag = true then} \)
15: \( \text{ac} \leftarrow \text{getAnotherSet(AC)} \) //call heuristic backtracking algorithm.
16: \( \text{end if} \)
17: \( \text{end while} \)
18: \( \text{end for} \)
19: \( \text{end procedure} \)
```

2.2.1. "Pruning" algorithm for simplification

We can use the set of atomic expression constraints(ACS) to represent the logical expression constraint. In this case, the relationship among variables that inside the atomic expression is ignored and treated as a whole. In this way, we can find the logical value combination of atomic expressions satisfying the logical expression constraints by establishing truth tables, and the results will be expressed by union sets.

For example, a simple logical expression constraint, \( LC = \{ Exp_1 \&\& Exp_2 \|\| Exp_3, F \} \), and the truth table of it is as shown in table 1.

From table 1, we can see that there are diversity and complexity in the combination of atomic expressions, especially when a logical expression constraint consists of multiple atomic expression constraints, the solution space will grow exponentially. And from table 1, we can get ACS = \( \{\{Exp_1, T\} \cap \{Exp_2, F\} \cap \{Exp_3, F\}\} \cup \{\{Exp_1, F\} \cap \{Exp_2, T\} \cap \{Exp_3, F\}\} \cup \{\{Exp_1, F\} \cap \{Exp_2, F\} \cap \{Exp_3, F\}\} \). In order to improve the efficiency of solving, the set of atomic expressions can be simplified. Simplification is based on the initial variable domain, and the atomic expression constraint that is always satisfied will be reduced by the "pruning" method, thus the unnecessary solving process is reduced. For example, in the ACS set, if it is certain that the initial variable domain of \( Exp_1 \) is
constant $T$, then the ACS set can be simplified to $\text{ACS} = \{\text{Exp}_2, F\} \cap \{\text{Exp}_3, F\}$, it can be seen from this that the "pruning" algorithm can greatly reduce the solution space of the constraint.

Table 1. Truth table.

| $\text{Exp}_1$ | $\text{Exp}_2$ | $\text{Exp}_3$ | LC |
|---------------|---------------|---------------|----|
| $T$           | $T$           | $T$           | $T$|
| $T$           | $T$           | $F$           | $T$|
| $T$           | $F$           | $T$           | $T$|
| $T$           | $F$           | $F$           | $F$|
| $F$           | $T$           | $T$           | $T$|
| $F$           | $T$           | $F$           | $F$|
| $F$           | $F$           | $T$           | $T$|
| $F$           | $F$           | $F$           | $F$|

2.2.2. Variable Level Determination Algorithm based on variable dependencies

According to the dependency between variables, the Variable Level Determination Algorithm (VLDA) sorts the ACS and solves the domain of the high-priority variable, so that the variable domain of the solution is more accurate, which is beneficial to the selection and generation of test cases.

Algorithm 2 Path-oriented logical expression constraint solving algorithm

Input: The set of atomic expression constraints, $P$;
Output: The set of variable; The set of atomic expression constraints, $SetP$;

1. Traverse all atomic expression constraints, set the highest priority $p_0$ to variables that directly depend on constraints, and the corresponding constraint set is $SetP_0$;
2. Set the variable $x$ that depends on the element in $P_0$ to priority $p_0$, and the corresponding constraint set is $SetP_1$, and the constraint does not belong to $SetP_0$;
3. Continue recursively solving the set of priority $P_n$, until all elements are present in the constraint set $SetP_n$;
4. The variables at the same level are solved in order;
5. return $P_n, SetP_n$;

For example, suppose the logical expression constraint is $\text{LC} = \{a < b \& \& a > 5 \& \& c > b, T\}$, the variable $a$, $b$ and $c$ are integer variables, and the initial domain of the variables are $(-\infty, +\infty)$. We can quickly get $\text{ACS} = \{a < b, T\} \cap \{a > 5, T\} \cap \{c > b, T\}$. According to the initial variable domain and ACS, if we call the interval algorithm to calculate each atomic expression in order from left to right, the result of solving ACS will be $\{a:(5, +\infty), b:(-\infty, +\infty), c:(-\infty, +\infty)\}$. Looking at the logical expression constraint, we can get dependencies between variables, the variable $c$ depends on the value of the variable $b$, the variable $b$ depends on the value of the variable $a$, and the variable $a$ depends on a constant. If we follow the dependency, we first solve $\{a > 5, T\}$, then solve $\{a < b, T\}$, and finally solve $\{c > b, T\}$, and the result will be $\{a:(5, +\infty), b:(6, +\infty), c:(7, +\infty)\}$. It can be seen that the algorithm can narrow the domain of variables, thus improving the efficiency of test case generation. The complete VLDA is defined as follows in Algorithm 2.

2.2.3. Heuristic backtracking algorithm for conflicts

After pre-processing, we can select an element of ACS and call the interval algorithm to solve the atomic expression to reduce the variable domain. However, there may be a conflict of solutions, that is, there is no value in the domain of variable that can satisfy the constraints. In this case, backtracking is required, and the new element of ACS will be re-selected to solve. One of the methods of selection is blind, that is, randomly choosing an element, which is simple and easy to implement, but it will easily lead to multiple conflicts; the other is heuristic, some heuristic information will be collected from the conflict. And according to this information, the results move in the direction most likely to reach the final state.
Figure 1. State at the time of constraint solving.

Suppose \( \text{LCS} = \{(x_2 > 5 && x_1 - x_2 > 10 | x_3 < -10 && x_1 - x_3 < 10) \& \& x_1 < 0, T \} \), and the initial variable domain is \( \{x_2: [8, +\infty]\} \). The ACS that satisfies LCS is represented by the path from the root node of the binary tree to the leaf node, as shown by state1 in figure 1. And after calling the pre-processing algorithm, the result will be shown in state2. As can be seen from the scale of the binary tree, the pre-processing algorithm greatly reduces the search space. Next, we can call the interval algorithm. Assuming that we select path1 when solving, start from the root node, when we calculate \( \{-0, T\} \), we find conflict, then we need to backtrack and choose \( \{-0, F\}\) in path2 to calculate. From the conflict, we can find that the relevant variable of the conflicting expression is \( \{5, T\} \), so the relevant expression on the path1 is \( \{5, T\}\) and \( \{-0, T\}\), the same as \( \{-0, T\}\) in path4 and path5, so we can exclude path1, path2, path4 and path5 at the time of backtracking.

3. Experiments

The method RDP (Recursive Descent Parsing) solves logical expression constraints recursively and adds all possible values of variables to the set of solutions which will lead the result to be \((-\infty, +\infty)\). And the domain of \((-\infty, +\infty)\) is meaningless to the interval reduction. In the experiment, we compared the reduction of the variable domain between the strategy of this paper and the RDP method. Statistics from three large projects indicate that the more complex the logical expression constraints are, the less the number of constraints is. And the most complex logical expression constraints consist of six atomic expression constraints. The experiment in this paper is set as follows, six variables, and these variables are arbitrarily combined into a set of linear atomic expressions by using mathematical operators and relational operators. We will select \(\pi (\pi \in [2,6])\) atomic expressions from the set and combine them into logical expressions by logical operators (\&\&,||,!), as a condition of the if statement in the program waiting for testing. [7]

The experiment tests 100 test programs and the test results of some functions are listed in table 2. As can be seen from experiment result, the strategy of this paper can support the logical expression constraint solving, reduce most of the variable domain to a limited range, and avoid the test case search failure.
Table 2. Comparison of the reduction of the variable domain.

| function   | LCS                                                                 | reduction of variable domain using the RDP method | reduction of variable domain using the strategy of this paper | times of searching test cases |
|------------|----------------------------------------------------------------------|--------------------------------------------------|---------------------------------------------------------------|-------------------------------|
| \(f_{01}\) | \(\{x_1 > -5 \& \& ! (x_2 > 10), F\}\)                           | \(\{x_1: (-\infty, +\infty), x_2: (-\infty, +\infty)\}\) | \(\{x_1: [-4, +\infty), x_2: [11, +\infty)\}\)             | 2                             |
| \(f_{02}\) | \(\{x_1 + x_2 < 10 \& \& x_1 > -5 \& \& x_2 + 2 \times x_3 < 25, F\}\) | \(\{x_1: (-\infty, +\infty), x_2: (-\infty, +\infty), x_3: (-\infty, +\infty)\}\) | \(\{x_1: [-4, +\infty), x_2: [-\infty, 13], x_3: [-6, +\infty)\}\) | 3                             |
| \(f_{03}\) | \(\{x_1 - x_2 > 10 \& \& x_2 - x_3 > 5 \& \& x_3 > 5 \& \& 10 \times x_4 + x_3 > 20 \& \& x_5! = 0, F\}\) | \(\{x_1: (-\infty, +\infty), x_2: (-\infty, +\infty), x_3: [5, +\infty), x_4: (-\infty, +\infty), x_5: (-\infty, -1)\}\) | \(\{x_1: [22, +\infty), x_2: [11, +\infty), x_3: [5, +\infty), x_4: (-\infty, +\infty), x_5: (-\infty, -1)\}\) | failed after searching 100 times |
| \(f_{04}\) | \(\{(x_2 > 5 \& \& x_1 - x_2 > 10) \& \& (x_4 == 0) \& \& x_3 + 2 \times x_4 = 10\) \& \& !(x_5 \times 2 < 4) \& \& x_5! = 0, F\}\) | \(\{x_1: (-\infty, +\infty), x_2: [6, +\infty), x_3: (-\infty, +\infty), x_4: (-\infty, +\infty), x_5: [2, +\infty), x_6: (-\infty, -1)\}\) | \(\{x_1: [17, +\infty), x_2: [6, +\infty), x_3: [10,10], x_4: [0,0), x_5: [2, +\infty), x_6: (-\infty, -1)\}\) | failed after searching 100 times |

The experiment also compares the time of generating test cases to illustrate the efficiency of the strategy. The experiment results show that 86 out of 100 functions can generate test cases more efficiently than the RDP method, and the time of solving the remaining functions is similar. The test results of some functions are listed in table 3. The result shows that the strategy of this paper has significantly improved the efficiency of generating test cases. This paper also analyses the cases that 14 of 100 functions where the test case generation time becomes longer after adding the logical expression constraint solving strategy. The main reasons are as follows. First, the test case generation radix is small, the difference is within a few milliseconds, and the machine state affects the efficiency. Second, when using the strategy of this paper, when there is a conflict that requires backtracking but still fails to solve, this will be accompanied by a large number of interval operations and waste more time.
### Table 3. Comparison of test case generation efficiency.

| function | Time of test case generation/ms | Time of test case generation/ms |
|----------|---------------------------------|---------------------------------|
|          | using the RDP method | using the strategy of this paper | using the RDP method | using the strategy of this paper |
| f06      | 2,007                          | 1,678                          | f53                  | 6,640                          | 3,536                          |
| f12      | 5,427                          | 1,691                          | f60                  | 5,936                          | 4,711                          |
| f19      | 6,644                          | 1,735                          | f64                  | 7,542                          | 4,120                          |
| f25      | 7,678                          | 1,804                          | f75                  | 5,501                          | 1,822                          |
| f28      | 3,733                          | 2,255                          | f76                  | 7,418                          | 4,309                          |
| f36      | 5,722                          | 3,687                          | f78                  | 6,793                          | 3,359                          |
| f41      | 5,899                          | 3,695                          | f92                  | 8,2558                         | 2,706                          |
| total    | --                             | --                             | --                   | 5,195                          | 41,108                         |

### 4. Conclusion

Based on the interval algorithm, this paper supports the logical expression constraint solving in path-oriented unit test and can successfully reduce the domain of the input variable. Meanwhile, the "pruning" and variable level determination algorithm improve the efficiency of test case generation. The experiment results show that the proposed logical expression constraint solving strategy can support complex logical expressions, and efficiency is also significantly improved compared with the RDP method. However, when there are many conflicts in the backtracking process, it will take a certain amount of time. How to make the conflict backtracking process faster and more efficient is a problem that needs to be studied in the future. In addition, the algorithm of this paper only supports linear constraints, and the solution to nonlinear constraints is also a problem to be solved in the future.

### Acknowledgments

This paper is supported in part by the National Natural Science Foundation of China under Grant No.U1736110 and No.61702044.

### References

[1] A. Gotlieb, B Botella, M Rueher, Automatic test data generation using constraint solving techniques[J], (ACM Press, 1998).
[2] R Dechter, D Frost, Backjump-based backtracking for constraint satisfaction problems[J], Artificial Intelligence, 147 (2002)
[3] R.A Demilli, A.J Offutt, Constraint-based automatic test data generation[J], Software Engineering IEEE Transactions on, 900 (1991).
[4] http://www.choco.solver.org/.
[5] G Alefeld, J Herzberger, Introduction to Interval Computations, (Academic Press, 1983).
[6] A.J Offutt, Z Jin, J Pan, The dynamic domain reduction procedure for test data generation[J], Software Practice & Experience (2015).
[7] https://github.com/byrcoderw/test/blob/master/experiment.c