International Journal of Future Generation Communication and Networking
Vol.7, No.3 (2014), pp.205-212
http://dx.doi.org/10.14257/ijfgcn.2014.7.3.19

A Method of Path Feasibility Judgment Based on Symbolic Execution and Range Analysis

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

[1]State Key Laboratory of Networking and SwitchingTechnology,Beijing University of Posts andTelecommunications,Beijing, China
[2]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[3]Liaoning Technical University, Huludao, China
wangyawen@bupt.edu.cn,lovelyjamie@yeah.net, laomao22311@126.com

Abstract

In program testing, the accurate information of path feasibility can improve the efficiency of static analysis. The dynamic judgment method of path feasibility needs to execute program, and the resultis usually not sound. On the basis of symbolic execution, this paper proposes a new static judgmentmethod, whichsimultaneouslycomputes two interval sets of each symbolic variable: possible value set and necessary value set. According to these range information, we can easily give the definitejudgment of a path: feasible, infeasible or uncertain. Experiment shows that the method is appropriate and efficient in case of the weakly relevant input.

Keywords:*path feasibility, symbolic execution, intervalcomputation, static analysis*

1. Introduction

The problem of path feasibilityjudgment, which is an important part of structure testing, has been studied since 1970s. The accurate information about path feasibility can improve the efficiency of static program analysis. Moreover, it is beneficial for testers to detect infeasible paths in early times, because generating test data for infeasible paths will consume a great deal of human and material resources in the subsequent dynamic testing stage.

Weyuker [1] has proved that it is an unsolvable problem to determine whether a program path is feasible or not. According to the study progress on path feasibility, there are three main strategies.

1) To select feasible paths based on fewer decision nodes, since the fewer predicates means the smaller probability of infeasible paths [2].

2)To judge infeasible paths dynamically. That is to say, to evaluate whether one path is feasible by the effort when generating test cases for the path [3-4].

3) To judge infeasible paths statically by analyzing the satisfication of path conditions or the effect of branch correlation [5-6].

However, because of the uncertainty of the checking results, the first two methods are just suitable for most of the program paths. Although the static method cannot determine the feasibility of all the paths definitely, its checking result is sound, which presents the path feasibility accurately, and it is much useful for the program testing.

In this paper, we propose a new judging method based on symbolic execution and static range analysis, which uses the extended interval arithmetic. As one of the static methods, it can accurately identify not only part of infeasible paths, but also part of the feasible paths, and this is the main contribution of this paper.

The remainder of this paper is organized as follows. Section 2 defines the possible value set and necessary value set of a variable in the condition expression and gives the

ISSN: 2233-7857 IJFGCN

method to compute them, Section 3 proposes the path feasibility judgment approach based on symbolic execution and interval analysis, and Section 4 shows our experiment results. Finally, Section 5 is our conclusion.

2. Possible Value Set and Necessary Value Set

The classic Interval Arithmetic is often said to have begun with Moore's book [7] in 1966 and used to solve the reliable boundary problem of numerical calculation in early times. Under Moore's direction and influence, general purpose interval analysis became available for use on physics, engineering, economy, and early computers.

We extended the theory of interval arithmetic in paper [8] and introduced several concepts, such as numeric interval-set, the Boolean arithmetic and the pointer arithmetic. In paper [8], we defined two interval value sets of variables:possible value set and necessary value set, to compute the ranges of variables in a condition expression easily and without changing the structure of abstract syntax tree.

Let *C* be a condition in program and let variables $v_0, v_1, ..., v_n$ be included in C, where *n* is the number of variables involved in the current scope.

Def 1. Let $E(v_i)$ be the interval value of $v_i$right before executing the condition C, and it is also called the current universal value set. Then, $C(v_0, v_1,..., v_n)$is a two-value function from $E(v_0)\times E(v_1)\times ...\times E(v_n)$to {0, 1}.

Def 2. Let $posbValue(C, v_i)$denotethe possible value set of variable v_i under the condition that C is true. Then,

$posbValue(C, v_i) =$
$\{x|\exists v_0\exists v_1 ... \exists v_{i-1}\exists v_{i+1} ... \exists v_n C(v_0, v_1, ..., v_{i-1}, x, v_{i+1}, ..., v_n)\}$.

Def 3. Let $necsValue(C, v_i)$ denote the necessary values set of variable v_i under the condition that C is true. Then,

$necsValue(C, v_i) =$
$\{x|\forall v_0\forall v_1 ... \forall v_{i-1}\forall v_{i+1} ... \forall v_n C(v_0, v_1, ..., v_{i-1}, x, v_{i+1}, ..., v_n)\}$.

According to the definitions stated above, we can obtain the following two properties.

Property 1.

$posbValue(\neg C, v_i)$
$= \{x|\exists v_0\exists v_1 ... \exists v_{i-1}\exists v_{i+1} ... \exists v_n \neg C(v_0, v_1, ..., v_{i-1}, x, v_{i+1}, ..., v_n)\}$
$= \{x|\neg\forall v_0\forall v_1 ... \forall v_{i-1}\forall v_{i+1} ... \forall v_n C(v_0, v_1, ..., v_{i-1}, x, v_{i+1}, ..., v_n)\}$
$= \sim necsValue(C, v_i)$
$= \mathrm{E}(v_i) - necsValue(C, v_i)$

Property 2.

$posbValue(C, v_i)$
$= \{x|\exists v_0\exists v_1 ... \exists v_{i-1}\exists v_{i+1} ... \exists v_n C(v_0, v_1, ..., v_{i-1}, x, v_{i+1}, ..., v_n)\}$
$= \{x|\neg\forall v_0\forall v_1 ... \forall v_{i-1}\forall v_{i+1} ... \forall v_n \neg C(v_0, v_1, ..., v_{i-1}, x, v_{i+1}, ..., v_n)\}$
$= \sim necsValue(\neg C, v_i)$
$= \mathrm{E}(v_i) - necsValue(\neg C, v_i)$

Let *VA* be the variable set consisting of all variables appeared in expression *A*, and let *VB* be the variable set of expression *B*. Thus, $VA = \{v_{a1}, v_{a2}, ..., v_{an}\}$, and $VB = \{v_{b1}, v_{b2}, ..., v_{bn}\}$.

(1) "*NOT*"$(\neg C)$

$posbValue(\neg C, x) = \mathrm{E}(x) - necsValue(C, x)$
$necsValue(\neg C, x) = \mathrm{E}(x) - \mathrm{posb}Value(C, x)$

(2) "*OR*"$(C = A \parallel B)$

The possible value set of a variable *x* in an *OR-expression* and its corresponding condition are shown in Table 1.

Table 1. Posb Value of a Variable x in an OR-Expression

posb$Value(C,x)$	Condition
posb$Value(A,x)$ ∪ posb$Value(B,x)$	$x \in VA \cap VB$
$E(x)$	$x \in VA \wedge x \notin VB \wedge \exists y(posbValue(B,y) \neq \Phi \wedge y \in VB)$
$E(x)$	$x \notin VA \wedge x \in VB \wedge \exists y(posbValue(A,y) \neq \Phi \wedge y \in VA)$
posbValue(A, x)	$x \in VA \wedge x \notin VB \wedge \forall y(posbValue(B,y) = \Phi \wedge y \in VB)$
posbValue(B, x)	$x \notin VA \wedge x \in VB \wedge \forall y(posbValue(A,y) = \Phi \wedge y \in VA)$

The necessary value set of a variable *x* in an *OR-expression* and its corresponding condition are shown in Table 2.

Table 2. Necs Value of a Variable x in an OR-Expression

necs$Value(C,x)$	Condition
necs$Value(A,x)$ ∪ necs$Value(B,x)$	$x \in VA \cap VB$
necs$Value(A,x)$	$x \in VA \wedge x \notin VB \wedge \forall y(necsValue(B,y) \neq \mathrm{E(y)} \wedge y \in VB)$
necs$Value(B,x)$	$x \notin VA \wedge x \in VB \wedge \forall y(necsValue(A,y) \neq E(y) \wedge y \in VA)$
E(x)	$x \in VA \wedge x \notin VB \wedge \exists y(necsValue(B,y) = E(y) \wedge y \in VB)$
E(x)	$x \notin VA \wedge x \in VB \wedge \exists y(necsValue(A,y) = E(y) \wedge y \in VA)$

(3) "*AND*" ($C = A$ &&B)

The possible value set of a variable *x* in an *AND-expression* and its corresponding condition are shown in Table 3.

Table 3. Posb Value of a Variable x in an AND-Expression

posb$Value(C,x)$	Condition
posb$Value(A,x)$ ∩ posb$Value(B,x)$	$x \in VA \cap VB$
posbValue(A, x)	$x \in VA \wedge x \notin VB \wedge \forall y(posbValue(B,y) \neq \Phi \wedge y \in VB)$
posbValue(B, x)	$x \notin VA \wedge x \in VB \wedge \forall y(posbValue(A,y) \neq \Phi \wedge y \in VA)$
Φ	$x \in VA \wedge x \notin VB \wedge \exists y(posbValue(B,y) = \Phi \wedge y \in VB)$
Φ	$x \notin VA \wedge x \in VB \wedge \exists y(posbValue(A,y) = \Phi \wedge y \in VA)$

The necessary value set of a variable *x* in an *AND-expression* and its corresponding condition are shown in Table 4.

Table 4. Necs Value of a Variable x in an AND-Expression

necs$Value(C,x)$	Condition
necs$Value(A,x)$ ∩ necs$Value(B,x)$	$x \in VA \cap VB$
Φ	$x \in VA \wedge x \notin VB \wedge \exists y(necsValue(B,y) \neq \mathrm{E(y)} \wedge y \in VB)$
Φ	$x \notin VA \wedge x \in VB \wedge \exists y(necsValue(A,y) \neq E(y) \wedge y \in VA)$
necs$Value(A,x)$	$x \in VA \wedge x \notin VB \wedge \forall y(necsValue(B,y) = E(y) \wedge y \in VB)$
necs$Value(B,x)$	$x \notin VA \wedge x \in VB \wedge \forall y(necsValue(A,y) = E(y) \wedge y \in VA)$

3. The Method of Path Feasibility Judgment

According to the definitions of possible value set and necessary value set, we can see that the former is an overestimation of the actual value, and the latter is an underestimation. For a path*p*, we assign a symbol for each input variable, and symbolically execute the path, meanwhile perform interval analysis for each symbol. Then, if the possible value set of some symbols*S1* is $\emptyset$, there exists some contradiction on *S1*and *p* is infeasible. Similarly, if the necessary value set of all the symbols isNOT $\emptyset$, *p* is feasible and the arbitrary combination of values from the necessary value sets can be a test case which drives the program run along the path *p* exactly.

Withthe consideration above, we give the following algorithm which calculates the possible value set and the necessary value set of each input variable. According to the output range information, we canjudgethe feasibility of the given path. Two definitions used in the algorithm follow.

Def 4. Symbolic Environment$X_{sym}{}^{\#}: \mathrm{Var} \rightarrow \mathrm{SymbExpr}$,where Var denotes the set of variables, and $\mathrm{SymbExp}$ is the set of symbolic expressions.

Def 5. Interval Range Environment $X_{itv}{}^{\#}: \mathrm{Symb} \rightarrow \mathrm{PosbInterval} \times \mathrm{NecsInterval}$, where Symb denotes the set of the symbols generated during the symbolic execution, $\mathrm{PosbInterval}$ is the set of possible value intervals, and $\mathrm{NecsInterval}$ is the set of necessary value intervals.

Algorithm 1: Path-Wise Symbolic Interval Computation

---

input：a program path *p*、the input variable set *vars*
output：the possible value set and the necessary value set of each input variable
procedurepathAnalysis(Path *p*, VarSet *vars*) {
for each*edge****in****p* {
$X_{itv}{}^{\#}$(*edge*)= ∅;
$X_{sym}{}^{\#}$(*edge*)= ∅;
}
out = *entry*.outEdge(); //the edgeto be analyzed in path *p*
foreach*v****in*** *vars***{**
updateSymbolEnv($X_{sym}{}^{\#}$(*out*), *v*, *sym***}**;
updateIntervalEnv($X_{itv}{}^{\#}$(*out*), *sym*, ⊤, ⊤); // to generate a new symbol *sym* whose interval is the upper bound ⊤
}
node = *edge*.headNode();
while (*node* != *exit*) **{**
Edge *in* = *node*.inEdge(); // the in-edge of the current node
Edge *out* = *node*.outEdge();//the out-edge of the current node
$X_{sym}{}^{\#}$(*out*)= $X_{sym}{}^{\#}$(*in*); $X_{itv}{}^{\#}$(*out*)= $X_{itv}{}^{\#}$(*in*); //to initialize the environment of the out-edge with the one of the in-edge
if (*node****in*****Calls) {** //function call statement
updateIntervalEnv($X_{itv}{}^{\#}$(*out*), *sym*, *sum*.retValue()); // to generate a new symbol *sym*,and *sum*is the function summary of the called function
}
case*node****in*****{**
Declarations: // declaration statement
for each*v****in****node***{**
updateIntervalEnv($X_{itv}{}^{\#}$(*out*), *sym*, ⊤);// to generate a new symbol *sym* whose interval is the upper bound ⊤
updateSymbolEnv($X_{sym}{}^{\#}$(*out*), *v*, *sym***}**;
}
Assignments: // assignment statement
Var*v* = *node*.id(); // the left-hand assigned variable
convert(*node*.expr(), *symExpr*); // to replace the right-hand expression with its corresponding symbolic expression *symExpr*
updateSymbolEnv($X_{sym}{}^{\#}$(*out*), v, *symExpr***}**;
Tests: //condition statement
convert(*node*.expr(),*symExpr*); // to replace the condition expression with its corresponding symbolic expression
for each*symVar****in****symExpr*
updateIntervalEnv($X_{itv}{}^{\#}$(*edge*), *symVar*, Interval(*symExpr*, *symVar*, *boolFlag*));
}


```
    node=out.headNode();
    }
}

procedure updateSymbolEnv(SymbolEnv se, Variable var, Expression exp) {
    if ( se.getExpr(var) !=null )
       se.delete(var);
    se = se∪{var→exp};
}

procedureupdateIntervalEnv(IntervalEnv ie, Symbol sym, Interval posb, Interval necs)
{
    if ( ie.getInterval(sym) !=null ) {
          if ( ie.getposbValue(sym) ==∅) {
          abort();
          p.setStatus("infeasible");
          }
          elseie.delete(sym);
    }
    ie = ie∪{sym→<posb, necs>};
}
```

Here we give an example to illustrate this algorithm. Figure 1 gives a code fragment of function f written in C Language. We choose two paths from the entry to the exit, *P1*: 1→2→3→4→5→**6→7**→10, Let *X* and *Y* be the symbols corresponding to the input variables *x* and *y*. After performing algorithm 1 on *P1*, we obtain the symbolic environment and the interval range environment of each edge, listed in Table 5. At the edge**6→7** of *P1*, posbValue(X) is ∅,then *P1* is identified as infeasible pathwhose execution cannot be caused by any set ofdata. Similarly, for another path *P2*: 1→2→3→4→5→**6→8→9**→10, at the last edge **9**→10, necsValue(X) is {[65, 90]}and necsValue(Y) is {(-∞,96]}, neither of them is ∅, *P2* is a feasible path. Selected data from necsValue(X) and necsValue(Y) arbitrarily, we can construct one possible test case for *P2*,{x=75, y=10}.

```
1: int f(int x, int y){
2:     if(x>64&&x<91)
3:          x+=32;
4:     if( y<x )
5:          y+=10;
6:     if (y>256)
7:          return 0;
8:     else
9:          return 1;
10: }
```

Figure 1. A Code Fragment of Function f

Table 5. The Symbolic Environment and Interval Range Environment on Path P1

Initialization: input vars={x, y}						
edge	$X_{sym}^{\#}$(edge)		$X_{itv}^{\#}$(edge)			
	symExpr(x)	symExpr(y)	posbValue(X)	necsValue(X)	posbValue(Y)	necsValue(Y)
1→2	X	Y	{(−∞, +∞)}	{(−∞, +∞)}	{(−∞, +∞)}	{(−∞, +∞)}
2→3	X	Y	{[65, 90]}	{[65, 90]}	{(−∞, +∞)}	{(−∞, +∞)}
3→4	X+32	Y	{[65, 90]}	{[65, 90]}	{(−∞, +∞)}	{(−∞, +∞)}
4→5	X+32	Y	{[65, 90]}	{[65, 90]}	{(−∞, 121]}	{(−∞, 96]}
5→6	X+32	Y+10	{[65, 90]}	{[65, 90]}	{(−∞, 121]}	{(−∞, 96]}
6→7	X+32	Y+10	{[65, 90]}	{[65, 90]}	∅	∅
7→10	-	-	-	-	-	-

4. Experiment

We selected several typical C programs as the experiment input. Table 6 lists the basic information, including the number of LOC(Line of Code), branches, parameters, and paths based on the branch coverage criterion. The experiment environment was set as follows: Intel Pentium 2.50 GHz CPU, 4.0 GB Memory and Windows 7Ultimate OS. Performing Algorithm 1 on each path respectively, we calculated the number of infeasible paths and feasible paths, the average time cost by every path, and the result is added in the last 3 columns. It can be seen that the number of paths identified definitely accounts for about 90.2%, and the average time is only 8.1ms.This approach is efficient and rapid for the general program.

Table 6. The Testing Result of 10 C Programs

Function name	Metric information				Result		
	LOC	Number of branches	Number of parameters	Number of paths	Number of infeasible paths	Number offeasible paths	Average time(μs)
bonus	29	10	1	6	0	6	18,684
days	33	17	3	52	25	27	14,684
division	26	6	1	5	2	2	4,919
equation	31	6	3	4	0	4	10,303
pingpang	17	6	3	4	0	0	3,178
prime	20	8	1	9	5	4	5,471
lsqrt	99	10	1	14	14	0	6,045
star	17	6	3	4	1	3	2,925
statistics	21	8	5	5	0	5	5,524
triangle	52	28	3	89	59	0	9,172

5. Conclusion

In static program analysis, it usually requires very difficult work to definitely judge the feasibility of a given path. This paper introduces a simple method of path feasibility judgment based on symbolic execution and extended interval arithmetic, and the checking result is sound. If the result tells that one path is feasible or infeasible, it is true with the fact. Of course, limited by the approximation of interval arithmetic, the method is effective for the functions with weakly relevant input, and there may exist quite many paths identified *uncert*ain. Our next work is to improve the precision of interval computation with complex expressions and operators. By this way, a significant accuracy enhancement of path feasibility identification will be achieved.

Acknowledgement

This paper is supported by the National Natural Science Foundation of China (No. 61202080), the National Grand Fundamental Research 863 Program of China (No. 2012AA011201), the Key Project of the National Natural Science Foundation of China (No. 91318301) and the Open Funding of State Key Laboratory of Computer Architecture (CARCH201201).

References

[1] E. J. Weyuker, "The Applicability of Program Schema Results to Programs", Int. J. Comput. Inf. Sci., vol. 8, **(1979)**, pp. 387-403.
[2] D. Yates and N. Malevris, "Reducing the Effects of Infeasible Paths In Branch Testing", ACM SIGSOFT Software Engineering Notes, vol.14, no. 8, **(1989)**, pp. 48-54.
[3] R. P. Pargas, M. J. Harrold and R. R. Peck, "Test data generation using geneticalgorithms", Journal of Software Testing, Verification and Reliablility, **(1999)** September, pp. 263-282.
[4] P. M. S. Bueno, M. Jino, "Identification of potentially infeasible program paths bymonitoring the search for test data", Proceedings ASE2000. The Fifteenth IEEE International Conference on Automated Software Engineering, **(2000)**, pp. 209-218.
[5] A. Goldberg, T. C. Wang and D. Zimmerman, "Applications of feasible path analysis toprogram testing", Proceedings of the 1994 International Symposium on SoftwareTesting and Analysis (ISSTA), **(1994)**, pp.80-94.
[6] F. Mueller and D. B. Whalley, "Avoiding conditional branches by code replication", Proceedings of the ACM SIGPLAN'95 Conference on Programming LanguageDesign and Implementation, SIGPLAN Notices, **(1995)**, pp. 56-66
[7] R. E. Moore, Interval Analysis. Englewood Cliffs, NJ: Prentice-Hall, **(1966)**.
[8] Y. Wang, Y. Gong and J. Chen, "An application of interval analysis in software static analysis", Proceedings of The 5th International Conference on Embedded and Ubiquitous Computing, Shanghai, China, **(2008)**, pp. 367-372.