Program Analysis: From Qualitative Analysis to Quantitative Analysis (NIER Track)*

Sheng Liu
State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences
lius@ios.ac.cn
Graduate University, Chinese Academy of Sciences

Jian Zhang
State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences
zj@ios.ac.cn

ABSTRACT

We propose to combine symbolic execution with volume computation to compute the exact execution frequency of program paths and branches. Given a path, we use symbolic execution to obtain the path condition which is a set of constraints; then we use volume computation to obtain the size of the solution space for the constraints. With such a methodology and supporting tools, we can decide which paths in a program are executed more often than the others. We can also generate certain test cases that are related to the execution frequency, e.g., those covering cold paths.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

General Terms

Algorithms

Keywords

program analysis, symbolic execution, execution probability

1. INTRODUCTION

The correctness of programs has always been a concern for computer scientists and software engineers. In recent years, there have been a lot of research works on symbolic execution [5] and constraint solving techniques, which can help assure program correctness. See for example, [3, 7, 8]. It usually works by collecting a set of constraints using symbolic execution techniques and then solving these constraints using constraint solving techniques. In most cases, the constraint solving phase usually returns a single solution to the constraints. Given this solution as the initial input data, the program will be executed along the given path. We name this kind of analysis as qualitative analysis.

In this paper, we will go one step further and study analysis methods that are based on the number of solutions to the constraints. Such a method is called quantitative analysis [9]. Technically, this study follows our previous works on qualitative analysis because it also uses symbolic execution techniques. But it generalizes previous works on symbolic execution, by replacing the constraint solving part with a new part called volume computation [6]. In this way, when a set of program path constraints is given, we not only can obtain a single solution that satisfies the program path constraints using the constraint solving techniques, but also can tell how many solutions there are that satisfy the program path constraints using the volume computation techniques. The number of solutions that can satisfy certain program path constraints is called the volume of these constraints. And the work of combining symbolic execution techniques and volume computation techniques is called quantitative program analysis, which is the focus of this paper.

Based on the volume computation method, we will also introduce some interesting applications of the quantitative program analysis. For example, we will show its applications in detecting hot program paths, volume-guided static program branch prediction, and volume-guided test case generation and selection. We think that the generalization of static program analysis from qualitative analysis to quantitative analysis will introduce a new spectrum of problems and applications.

2. SYMBOLIC EXECUTION

Usually a program can be represented by a (control) flow graph which is a directed graph. In general, there are many or an infinite number of paths. A path is executable/feasible if there are some initial input values for the variables that can drive the program to be executed along that path. Otherwise, the path is unexecutable/infeasible.

A basic approach to path feasibility analysis is to compute the path condition (which is a set of constraints over the input variables), and decide whether it is satisfiable or not. The program path is feasible if and only if the path condition is satisfiable. So the path feasibility analysis problem can be reduced to a constraint solving problem.

*This work is supported in part by the National Natural Science Foundation of China (Grant No. 61070039) and the High-Tech (863) program of China (Grant No. 2009AA01Z148).

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

To obtain the path condition, we can use the symbolic execution technique, which has been studied by various researchers for more than 30 years. Its basic idea is very simple, namely, to "execute" a program using symbolic values for variables. The actual execution of a program requires concrete values (e.g., $x == 2$, $y == 8$) as the input data. In contrast, symbolic execution uses symbols as values of variables. Suppose there is an assignment statement like $z = 2x+y$. During concrete execution, the statement assigns a new value like 12 to the variable x. But with symbolic execution, the new value is typically some symbolic expression like $2a_0 + b_0$.

Once a path condition is obtained by symbolic execution, it may be handled by constraint solving techniques. As a simple example, a constraint solver can decide that the following constraint is satisfiable:

$$(n >= 10) \wedge ((n < 4) \vee (n > 22))$$

A solution can also be found, e.g., $n = 23$. This may serve as the initial input value in a test case.

For the past 10 years, there have been quite some works on static analysis and test data generation which are based on symbolic execution and constraint solving. One may also combine symbolic execution with concrete execution [7].

When a path condition is satisfiable (and the program path is executable), we may further ask how many input data values satisfy the path condition. Intuitively, the more values satisfy the condition, the more often the path can be executed. This will be discussed in the next section.

3. VOLUME COMPUTATION

As mentioned above, solving a constrained problem usually means finding one solution or reporting that there is no solution at all. For most applications, doing this is enough; but for others we would also like to know the total number of solutions. Knowledge of the number of solutions will give a new perspective on the problem. It conveys more information about the problem. In mathematics, given a set of linear inequalities which constitute a bounded geometric object, a solution to it is only one point in the geometric object while counting the solutions corresponds to computing its volume, which is also a very important problem. See Fig. 1. As for logic formulas, counting the solutions to a propositional logic formula, namely model counting, is an important notation related to approximate reasoning in artificial intelligence. As for program analysis, since each program branch predicate is also represented as the Boolean combination of numeric constraints, if we have the knowledge of the size of the solution space, namely volume, of such predicates we can also do lots of analysis works based on these quantitative information.

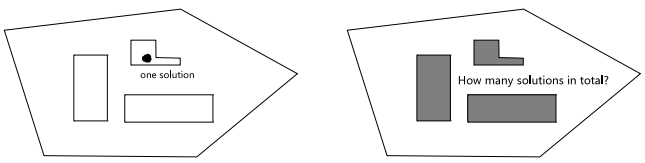


Figure 1: Constraint Solving vs Volume Computation and Model Counting

In [6], we presented several approaches for computing the size/volume of the solution space, given a set of constraints, where each constraint is a Boolean combination of linear arithmetic constraints. As an example, consider the formula

$$(((y+2x < 10) \rightarrow (30 < y)) \vee (x \leq 80)) \wedge ((30 < y) \rightarrow \neg(x > 5) \wedge (x \leq 80))$$

We first introduce a Boolean variable for each linear inequality and transform the formula to the following:

$$((b_1 \rightarrow b_2) \vee b_4) \wedge (b_2 \rightarrow \neg b_3 \wedge b_4)$$

where

$$\begin{cases} b_1 \equiv (y + 2x < 10) \\ b_2 \equiv (30 < y) \\ b_3 \equiv (x > 5) \\ b_4 \equiv (x \leq 80) \end{cases}$$

With our volume computation tool, we can obtain the volume of the above constraints. For more details of our volume computation methods and tool, please refer to [6].

4. QUANTITATIVE PROGRAM ANALYSIS

In this section, we briefly introduce some quantitative program analysis methods based on volume computation and solution counting.

4.1 Hot Path Detection

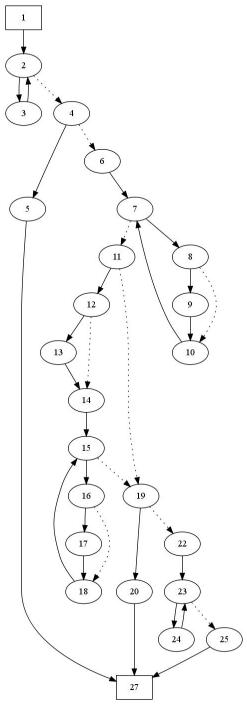
Empirical observation shows that most of the execution time of the program is spent on a small percentage of program paths. This makes the identification of hot paths of a given program an important research problem and a hot research topic. A most recent work on this is given by Buse and Weimer [2] who proposed a method for *estimating* path execution frequency. They give a statistical model, which is based on *syntactic* features of the program's source code. In contrast, with the *semantic* information in the program paths, we can identify hot paths precisely by calculating the *exact* execution probability of each program path.

Take the program *getop()* from [4] as an example. First of all, the control flow graph of the program, which is demonstrated in Fig. 2, is generated. Each node in the graph represents a conditional expression or a block of statements, as shown on the right hand side of Fig. 2. At a branch node, the dotted arrow represents the `true` branch, and the solid arrow represents the `false` branch. Then many paths from the program are obtained from the graph, and the path condition for each path is computed using the symbolic execution techniques. At last, the volume of each path condition is calculated with our tool.

In the experiment, we assume that each character variable is an integer, taking values from the range $[0, 255]$. We can see that some paths have a larger probability of being executed, as compared with other paths. For example, see the two paths: $Path1$ is $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 27$ and $Path2$ is $1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 11 \rightarrow 19 \rightarrow 20 \rightarrow 27$. With our tool, we found that the execution probability of $Path1$ is about 0.945 and that of $Path2$ is about 0.021. So $Path1$ can be regarded as a hot path.

4.2 Branch Prediction

Branch prediction is an important problem in compiler optimization. Good branch predictor can identify frequently executed regions and thus aid in scheduling instructions [1]. In the literature of branch prediction, although both the profile-based branch predictor and the program-based branch predictor are efficient to some extent, neither of them are precise enough because they do not take the semantic information of the program into account. However, armed with



```
1:  c=getchar();
2:  !(((c==32||c==9)||c==10))
3:  c=getchar();
4:  !((c!=46&&(c<48||c>57)))
5:  RETURN: c;
6:  s[0]=c; c=getchar(); i=1;
7:  !((c>=48 && c<=57))
8:  !(i<lim)
9:  s[i]=c;
10: c=getchar(); _temp_var0=i;
    i=i+1;
11: !(c==46)
12: !(i<lim)
13: s[i]=c;
14: c=getchar(); _temp_var1=i;
    i=i+1;
15: !((c>=48 && c<=57))
16: !(i<lim)
17: s[i]=c;
18: c=getchar(); _temp_var2=i;
    i=i+1;
19: !(i<lim)
20: s[i]=0;
    RETURN: 1000;
22: NOP
23: !((c!=10 && c!=-1))
24: c=getchar();
25: _temp_var3=lim-1;
    s[_temp_var3]=0;
    RETURN: 9999;
27: END
```

Figure 2: getop()

the symbolic execution and volume computation tools we can carry out the branch prediction task in a fine-grained manner precisely. For example, in the following program fragment, at branch node 3, since x is between 10 and 90, by volume computation we can obtain the accurate execution probability of the `true` branch, which is 75%. So the `true` branch is generally "busier" than the `false` branch. However, if we replace the number 100 in line 1 by 50, we can learn the change of "busy" branch immediately as compared with other methods.

```
1:  if((x <= 100) && (x >= 20)){
2:        x = x - 10;
3:        if (x > 30)
4:             ... //true branch
5:        else
6:             ... //false branch
7:  }
```

Now let us look at the example in Fig. 2 again. Each branch node of the graph is labeled with a branch predicate. For instance branch node 2 is labeled with

```
!(((c==32 || c==9) || c==10))
```

When carrying out the branch prediction work, we execute the volume computation routine at each branch node during the symbolic execution process of the program, so as to obtain the execution probability of the partial path up to that branch node. Suppose that the program is executed along the partial path $1 \to 2 \to 4$ and we would like to predict which branch of node 4 will be taken. Firstly we carry out the symbolic execution and volume computation task along the path $1 \to 2 \to 4$ and obtain the execution probability of the partial path, which is 0.988. (Each constraint collected along the path follows the symbol '@' as described below).

```
1:  c=getchar();
2:  @ !(((c==32 || c==9) || c==10))
```

To predict which branch of node 4 will be executed more frequently, we go on executing the program along the partial path $1 \to 2 \to 4$ and suppose that we reach node 5. The path constraints are listed below.

```
1:  c=getchar();
2:  @ !(((c==32 || c==9) || c==10))
!4: @ ((c!=46 && (c<48 || c>57)))
```

After symbolic execution and volume computation, we can also compute the execution probability of the partial path $1 \to 2 \to 4 \to 5$. The probability of node 4's leading to node 5 is the conditional probability of the event that the program is executed along $1 \to 2 \to 4 \to 5$ assuming the event that the program is executed along $1 \to 2 \to 4$. With the execution probabilities of the two partial paths, we can compute the conditional probability. We call this kind of branch prediction volume-guided branch prediction. As for this instance, the execution probability of the partial path $1 \to 2 \to 4 \to 5$ is 0.945. So the conditional probability is $0.945/0.988 = 0.956$, greater than 50%. It shows that at branch node 4 the program will take the `false` branch and head for node 5 with the probability 0.956.

We want to mention that volume-guided branch prediction can also help to enhance the hot path detection problem. In the above section, when detecting hot paths, we have to generate all the paths of the program from the flow graph exhaustively first. This has some drawbacks. For example, according to empirical observation, only a small percentage of program paths are hot paths, but there can be a huge number of program paths in total, especially when loops are considered. To find this small percentage of hot paths from possibly exponential number of paths, we can make use of the volume computation routine during the program path generation process to prune some unnecessary paths. One choice is to set a fixed parameter p, say $p = 20\%$, for each branch node. If the branch prediction ratio of a branch node is smaller than the fixed parameter, we prune one of the two subtrees under the branch node. Another choice is to set a fixed parameter h for the hot path detection problem. It means that all the paths with execution probability larger than h are hot paths. In each branch node, we compute the execution probability of the partial path up to this branch node. If the execution probability is smaller than h, then both subtrees of the branch node are pruned. In this way, after pruning unnecessary branches, all the leaf nodes of the remaining binary search tree are all the hot paths that we want.

Further, the method can also help to improve loop handling. When loops are taken into account, in previous works the path generation process is based on the syntax of the program and each loop is unfolded K times with a given K, which is common in the literature. In this case, we can hardly know whether this K is big enough in the context of hot path detection. But if we carry out the volume computation task in every branch node, we can obtain more semantic information in addition to syntactic information. For instance, when we interleave the loop unfolding process with the volume computation routine, if at certain unfolding phase the volume of the path condition is already below certain threshold, we immediately have the knowledge that the current partial path can not induce any hot paths. Then we

can stop the unfolding process of the loop. In other words, volume-guided branch prediction can also be used to predict the unfolding parameter K of each loop, which further helps to alleviate the path explosion problem in path generation.

4.3 Test Case Selection

A challenging problem in software testing is the design of good test suites. An important class of testing methods is called path oriented testing. It tries to find input test data such that every selected path is executed with the help of symbolic execution. Based on symbolic execution and constraint solving, in the past several years we have already done some work on test case generation [8]. When we analyze the program using the symbolic execution method, each program path will introduce a constraint. In previous works, we just solve the constraint to obtain a solution which is returned as one test case for this program path. This kind of test case generation ensures that each of these paths is feasible and all of the program paths altogether satisfy some testing criteria like branch coverage or statement coverage criterion. But now that we have the volume information of the constraint, we can do some more useful things.

A key observation is that all the program paths are not the same in terms of "volume". In general, the volume of the program path condition is an important measure for the study of each program path's capacity and fraction of coverage. In path-based test data generation, one may ask what is the difference between all these test cases? Which one is more important?

We may look at the problem from a different perspective. In some applications, it is the volume of the path condition, from which the test case is derived, that counts. For example, in some circumstances, a test case from a path with smaller volume ("cold" path) may be less representative than a test case from a path with bigger volume ("hot" path). But in some other circumstances, paths with smaller volume may be more important. It depends on which kinds of paths or program behaviors you like to inspect. For instance, sometimes we want to test some very unfrequented paths or some rare program behaviors, then in this case paths with smaller volume will certainly have higher priority to be tested. On the other hand, some other applications may pursue the uniformity of all the test cases, in this circumstance paths with bigger volume should produce more test cases than those with smaller volume. So with volume information of each path, we can do many interesting and useful things.

5. EXPERIENCES AND OPEN ISSUES

Previously we developed a symbolic execution tool [10] which generates path conditions for a given path, and tools for generating paths from a program [8]. More recently, we implemented an automatic tool for computing the volume (solution density) of complex constraints [6]. We have experimented with some benchmarks such as `getop()`, TCAS and some sorting/searching programs.

Our experiences show that the proposed approach works well when the program (path) can be executed symbolically. It can tell us how many input data will drive the program to be executed along a given path. As for the volume-guided test case generation, we have not experimented with it thoroughly.

It is still challenging to deal with complicated features such as pointers and function calls. (Of course, a simple way of dealing with functions is just inlining the function body.) Another issue is the tradeoff between the complexity of volume computation and the precision of the analysis results. Also, for some paths, the path condition leads to a volume of 0. How can we compare such paths? This issue needs to be investigated further.

To use the approach wisely, we may also need some domain knowledge. Currently, we assume that each variable can take a value from some bounded domain and that each value can be taken randomly. This may not be the case. For instance, in the TCAS example, how can we decide the upper bound of altitudes? Will it be 1000, 10000 or some larger value? In practice, it is quite unlikely for the variable to take a very big value. Can we take the probability distributions of the variables into account?

6. CONCLUSIONS

Symbolic execution is an important method for program analysis and testing. When combined with efficient constraint solving techniques, it can be used for accurate static analysis, bug finding and test data generation. In this paper, we proposed to combine symbolic execution with volume computation to compute the exact execution frequency of program paths and branches. With this methodology and existing tools, we are able to decide which paths in a program are executed more often than the others. We can also generate certain test cases that are related to the execution frequency, e.g., those covering cold paths. Preliminary experiences are encouraging. Compared with previous approaches, our approach offers finer analysis and more accurate results.

7. REFERENCES

[1] T. Ball and J. R. Larus. Branch prediction for free. In *Proc. of PLDI'93*, pages 300–313. ACM, 1993.

[2] R. P. L. Buse and W. R. Weimer. The road not taken: Estimating path execution frequency statically. In *Proc. of ICSE'09*, pages 144–154, 2009.

[3] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *Proc. of the PLDI'08*, pages 281–292. ACM, 2008.

[4] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[5] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, 1976.

[6] F. Ma, S. Liu, and J. Zhang. Volume computation for boolean combination of linear arithmetic constraints. In *Proc. of CADE-22, LNCS 5663*, pages 453–468. Springer-Verlag, 2009.

[7] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. of ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[8] Z. Xu and J. Zhang. A test data generation tool for unit testing of C programs. In *Proc. of QSIC*, pages 107–116, 2006.

[9] J. Zhang. Quantitative analysis of symbolic execution. *presented at COMPSAC'04*, 2004.

[10] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.