Constraint Solving and Symbolic Execution

Jian Zhang

Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing 100080, China
Email: zj@ios.ac.cn

1 Introduction

For many decades, the correctness of programs has been a concern for computer scientists and software engineers. At present, it is still not easy to ensure the correctness of nontrivial programs, although many researchers have made various attempts in this direction.

Recently, the Verifying Compiler is proposed as a grand challenge in computing research [3]. But its goal can be achieved incrementally. The following is quoted from Hoare (page 68 of [3]):

> The progress of the project can be assessed by the number of lines of code that have been verified, and the level of annotation and verification that has been achieved. The relevant levels of annotations are: structural integrity, partial functional specification, total specification. The relevant levels of verification are: by testing, by human proof, by machine assistance, and fully automatic.

For program verification to become mainstream technology in software engineering, we need to convince programmers that the benefit will outweigh the "investment". Obviously, highly efficient and easy-to-use tools are necessary. In this paper, we briefly describe and evaluate a path-oriented approach to partial program verification, which is based on Constraint Satisfaction and Symbolic Execution (CoSEx). It can be regarded as something between testing and full-scale verification. We think that the approach is quite appealing and can serve as the basis of powerful tools. For some programs, their correctness can be proved automatically by the tools; and for many other programs, if they have bugs, the bugs can be found automatically.

2 Semantic Path-oriented Analysis

As we know, a program can usually be represented by some kind of directed graph, e.g., control-flow graph, (extended) finite-state machine. From such a graph, one can generate many paths, each of which starts with the entry of the program (or module).

Path-oriented testing is a common testing strategy. With this kind of strategy, one tries to examine the program's paths one by one. But in general, a non-trivial program has too many (or an infinite number of) paths, and it is impossible to examine all of them within a reasonable amount of time. Thus the concept of "basis paths" was proposed. Such paths are expected to be representatives of the set of all paths.

In the software engineering literature, most testing techniques are *syntactic*, in that they tend to neglect the exact meaning of statements and conditional expressions in the program. For example, one may consider the `def-use` relationship (i.e., where a variable is defined/modified, and where it is used), but not consider the way the variable is modified. This kind of abstraction is necessary if the techniques and tools are used on large-scale programs. But the price to pay is the loss of accuracy and expressiveness. For example, a typical problem with most path-oriented testing methods is that many paths (in the program's control flow graph) turn out to be unexecutable. Static analysis tools often generate false alarms. And so on.

Alternatively, we can work on *semantic* path-oriented program analysis. That is, we take the meanings of the statements and conditional expressions into consideration. Under certain restrictions, we can perform some kind of partial verification on many programs, fully automatically. This is done in the following several steps:

(1) Annotate the program with appropriate assertions (preconditions or post-conditions).
(2) Generate a set of paths from the program's graphical representation.
(3) For each path, decide whether it is executable.

We may attach the negation of a correctness property at the end of the program. If input data are found such that the program reaches the end while satisfying all the assertions, the program is buggy.

In the third step, we first obtain a set of constraints (called the *path condition*), such that it is satisfiable if and only if the path is executable. This can be done through symbolic execution.

Symbolic execution [4] is a technique for testing and verification. During the execution, each variable's value is a symbolic expression, in terms of the initial values of the input variables. For example, suppose we have the input variable a and b, whose initial values are denoted by a_0 and b_0, respectively. Then, after the assignment $x = a + 2b$, the value of the variable x will be $(a_0 + 2 * b_0)$.

After executing a path symbolically, we get the path condition, which is a relational expression describing the constraints on the initial values of the input variables, e.g., $a_0 + 2 * b_0 > 4$. Given any vector of values satisfying the path condition, the program will be executed along the path. Thus the path condition represents a set of input data. It is a subset of the input space, yet it is usually infinite. One symbolic execution may correspond to many real executions.

The satisfiability of the path condition can be decided using various techniques, such as decision procedures, theorem proving, constraint solving, etc.

We should note that there is some trade-off between the expressiveness of the language and the difficulty of deciding path executability. In the following, we list several forms of the constraints and the hardness of the associated decision problem:

- Boolean formulas: decidable, NP-hard
- linear constraints over rationals: decidable, linear-time
- linear constraints over rationals and integers: decidable, NP-hard
- non-linear constraints over integers: undecidable

In [5, 7], a prototype toolkit is described, which uses symbolic execution and constraint solving techniques. It analyzes a subset of C programs statically. Non-linear arithmetic is not allowed, but logical operators can be used in the program. Pointers and structures are considered in [6]. Ordinary assertions (in C programs) are accepted, but not quantified formulas. This restriction reduces the complexity of the decision algorithm. Moreover, assertions are actively used by many programmers for various purposes [2].

The CoSEx approach performs path-wise analysis, and it analyzes each path accurately (under reasonable assumptions of the syntax of the path). It can be used to verify the correctness of certain programs, e.g., bubble sorting program when the size of the input array is a fixed constant. Such a program has a finite number of (symbolic) execution paths. However, most programs have an infinite number of paths, and the approach can only be used to find bugs (if any).

3 Comparison with Related Approaches

We think that there are several factors to consider when comparing different approaches. These factors include:

- applicability (e.g., the restriction to finite-state programs)
- power or preciseness (e.g., full verification, partial verification, bug finding with or without false alarms)
- the user's investment (e.g., writing a lot of lemmas, or just writing the precondition and the postcondition)

Compared with traditional testing and static analysis techniques, the CoSEx approach does not scale up to large programs (such as programs with millions of lines of code), yet it can provide the user with accurate analysis results. False alarms are eliminated in most cases.

Compared with model checking, the CoSEx approach does not require the program to have finite number of states. Although there have been some extensions to model checking so that it can be used to verify infinite-state systems, their effectiveness has yet to be seen.

Compared with theorem proving, the CoSEx approach is more automatic, but the properties it can prove are not so general. It is more advantageous to use the approach on buggy programs.

An abstract interpretation-based static program analyzer like ASTRÉE [1] considers a superset of the possible program executions, while our path-oriented approach considers a subset of all the executions, because only a finite number of paths are analyzed. But it may avoid false alarms.

An Example

In [8], 5 modern static analysis tools (including Splint and PolySpace) are evaluated using a number of nontrivial model programs which contain buffer overflows. It is found that, in some cases, some tools are silent, while other tools can detect the vulnerabilities and signal many false alarms.

Two false alarm examples ("aia2" and "inp") are given in Fig. 5 and Fig. 6 of [8]. We have tried our toolkit on the first example, since the second one has complicated expressions which are beyond the scope of the tools.

In the example "aia2", there are two arrays (`x` and `y`), and there is an assignment "`y[x[i]] = i`" (line 8). Although one element of the array `x` has the value `(-1)`, that value is never used to index into `y`. Thus there is actually no underflow. Using our tool ePAT (which is an extension of PAT [5]), we can check that this is indeed the case. We just run ePAT twice, each time attaching one the following two assertions to the statements before line 8:

```
@(x[i] < 0);
@(x[i] >= 2);
```

Here `@` denotes an assertion. It is found that neither of the two extended paths is executable. Thus the index expression `x[i]` is within the bound. (The array `y` is of size 2.)

Other similar tools like PREfix are not able to perform this kind of analysis. They are aimed at analyzing much larger programs.

4 Concluding Remarks

Up to now, only a few programmers have used program verification technology in developing nontrivial software. Wider use of the technology calls for powerful and efficient supporting tools, although education is also quite important.

An approach is outlined and evaluated in this paper. It is based on the analysis of program paths, and the analysis involves detailed semantic information. Symbolic execution and constraint solving techniques are used, which are automatic and accurate. Hopefully this approach will lead to rewarding tools for average programmers.

There are still other difficulties and challenging problems, for example, how to deal with procedures/functions. In the near future, we expect that the approach is applicable to small or medium-sized programs or key modules in the software system. While most other techniques try to scale up to large programs, we are more interested in scalability in the expressiveness of the input language. We think that the accurate analysis of small programs is also very important, and it can be complementary to other verification/analysis approaches.

References

1. Patrick Cousot *et al.* The ASTRÉE analyzer, *Proc. 14th European Symposium on Programming (ESOP 2005)*, LNCS 3444, Springer, 21–30, 2005.
2. C.A.R. Hoare, Assertions in modern software engineering practice, Keynote address, *26th Int'l Computer Software and Applications Conf.*, Oxford, England, Aug. 2002.
3. Tony Hoare, The verifying compiler: A grand challenge for computing research, *J. of the ACM*, 50(1): 63–69, 2003.
4. J.C. King, Symbolic execution and testing, *Comm. of the ACM*, 19(7): 385–394, 1976.
5. Jian Zhang and Xiaoxu Wang, A constraint solver and its application to path feasibility analysis, *Int'l J. of Software Engineering and Knowledge Engineering*, 11(2): 139–156, 2001.
6. Jian Zhang, Symbolic execution of program paths involving pointer and structure variables, *Proc. of the 4th Int'l Conf. on Quality Software (QSIC)*, IEEE Computer Society Press, 2004.
7. Jian Zhang, Chen Xu and Xiaoliang Wang, Path-oriented test data generation using symbolic execution and constraint solving techniques, *Proc. 2nd Int'l Conf. on Software Engineering and Formal Methods*, IEEE Computer Society Press, 242–250, 2004.
8. Misha Zitser, Richard Lippmann and Tim Leek, Testing static analysis tools using exploitable buffer overflows from open source code, *Proc. of the 12th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, 97–106, 2004.