Constraint Solving and Symbolic Execution*

Jian Zhang

State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
P.O. Box 8718, Beijing 100080, China

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 [7]. But its goal can be achieved incrementally. The following is quoted from Hoare (page 68 of [7]):

> 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. There are many automatic and efficient tools for testing and program analysis. But they still have some weaknesses.

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 how the variable is modified. In the programming language literature, many program analysis techniques focus on certain aspects of the programs. For example, pointer analysis algorithms typically neglect the values of non-pointer variables. This kind of abstraction is necessary for scalability. And most researchers would like to demonstrate that their techniques and tools can be used on large-scale programs. But the price to pay is the loss of accuracy and expressiveness. In fact, it is often very difficult to generate a good test suite for white-box testing, in which every test case is executable/usable. Similarly, it is hard to avoid false alarms in static analysis.

* Supported in part by the National Natural Science Foundation of China (NSFC).

We believe that it is worthwhile and feasible to analyze programs and specifications *accurately* and *automatically*. In this paper, we briefly describe and evaluate a path-oriented approach to (partial) program verification and testing, which is based on Constraint Satisfaction and Symbolic Execution (CoSEx). We think that the approach is quite appealing and can serve as the basis of powerful tools. Although the basic ideas have been known for a long time [1, 8, 2], serious efforts are needed to demonstrate the full power of the approach.

2 Path-oriented Analysis Based on Symbolic Execution

As we know, a program can usually be represented by some kind of directed graph, e.g., control flow graph, extended finite-state machine (EFSM). 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. Another way to get around the problem is to restrict the number of times each loop is executed. For example, we may consider just two cases: the loop body is not executed; the loop body is executed once.

Anyway, we assume that only a finite number of paths in the program are examined. But even under this assumption, automated verification and test data generation are still difficult. In software testing, a severe problem is that many program paths generated from the control flow graph are *non-executable* or *infeasible*. A path is *executable* (or *feasible*) if there are input data such that the program is executed along that path. Thus it is interesting to analyze each path accurately. To do this, we have to consider the full semantic information in the statements and conditional expressions.

Path-oriented analysis can also be used for some kind of partial verification. We may verify a program in the following steps:

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

In the first step, we attach the negation of a correctness property at the end of the program. If some path is executable, the program is found to be buggy.

The third step is crucial. To decide the feasibility of a path, we can first obtain a set of constraints (called the *path condition*), such that it is satisfiable if and only if the path is feasible.

Symbolic Execution

For a program path, the path condition can be obtained through symbolic execution [1, 8], which is a well-known 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. Typically, one symbolic execution corresponds 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.

Constraint solving

Constraint satisfaction problems [9] have been studied extensively in the artificial intelligence community. Informally speaking, such a problem consists of a set of variables, each of which may take a value from some domain. In addition, there are some constraints defined on the variables. Solving the problem means finding a value for each variable, such that all the constraints hold.

Obviously, the class of constraint satisfaction problems is quite general. Many problems fit into this framework, such as graph coloring and SAT (i.e., checking an arbitrary set of propositional clauses for satisfiability). For constraints having special forms, usually there are special methods for solving them. For instance, DPLL is a famous algorithm for solving SAT, and the simplex algorithm is very effective for solving linear arithmetic constraints.

We should note that there is some trade-off between the expressiveness of the constraint language and the difficulty of deciding the satisfiability. 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

Similarly, the more expressive a programming language is, the more difficult the analysis will be.

In [11, 10, 12], a prototype toolkit is described, which uses symbolic execution and constraint solving techniques. We call the toolkit SPAR. It analyzes a subset of C programs statically. Non-linear arithmetic is not allowed, but logical operators can be used in the program. 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 [6]. The constraint solving algorithm [11] is essentially a combination of linear programming and SAT solving. In contrast, earlier works like [1] typically uses linear programming only.

3 Comparison with Related Approaches

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

- generality or applicability (e.g., the restriction to finite-state programs)
- accuracy or preciseness (e.g., full verification, partial verification, finding certain bugs, generating few/many false alarms)
- degree of automation and efficiency (e.g., exponential or double exponential complexity)
- the user's investment (e.g., writing a lot of lemmas, or just writing the precondition and the postcondition, or giving no annotation)

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).

Compared with traditional testing and static analysis techniques, the CoSEx approach can provide the user with accurate analysis results. False alarms are eliminated in most cases. However, we do not think it will scale up to large programs (such as programs with a million lines of code) in the near future.

Compared with model checking, the CoSEx approach does not require the program to have a 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 effective to use the approach on buggy programs.

An abstract interpretation-based static program analyzer like ASTRÉE [3] considers a superset of the possible program executions, while the approach outlined in this paper considers a subset of all the executions, because only a finite number of paths are analyzed. But it can avoid many false alarms.

ESC/Java [5] and its successors are impressive static checking tools which can perform similar analysis on Java programs. But the underlying theorem prover, Simplify [4], accepts more expressive formulas which may contain quantifiers.

An Example

In [13], 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 [13]. 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 [11]), we can check that this is indeed the case. Since the array `y` is of size 2, we run ePAT twice, each time attaching one of the following two assertions to the statements before line 8: `@(x[i] < 0);` `@(x[i] >= 2);` Here `@` denotes an assertion. So we get two extended paths, one of which is the following:

```
int i;  int x[3], y[2];
{
  x[0] = 1;    i = 0;
 @(i < 3);
  x[i] = i-1;  i = i+1;
 @(i < 3);
  x[i] = i-1;  i = i+1;
 @(i < 3);
  x[i] = i-1;  i = i+1;
 @!(i < 3);
  i = 1;
 @(i < 3);
 @(x[i] < 0);
}
```

It is found that neither of the paths is executable. Thus the index expression `x[i]` is within the bound, and we have shown that the alarm is false.

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. The analysis involves detailed semantic information, and uses symbolic execution and constraint solving techniques which are automatic and accurate. Hopefully this approach will lead to rewarding tools for average programmers. With such tools, we should be able to

- verify – or find bugs in – certain programs (like `bubble_sort`, where the size of the input array is a fixed positive integer)
- check the error messages produced by other static analyzers, to eliminate some false alarms
- automate an important part of unit testing, i.e., generating test cases (input data) for the program
- generate test cases for black-box testing or model-based testing, if a proper specification (like EFSM) is provided.

We can also perform other kinds of analysis which are not so related to the correctness of programs.

In summary, we think that it is very important to analyze programs and specifications accurately and automatically. Such an analysis may use symbolic execution and constraint solving techniques. It can be complementary to other verification/analysis approaches. In the near future, we expect that the approach is applicable to small or medium-sized programs or key modules in software systems. While many other techniques try to scale up to large programs, we are more interested in scalability in the expressiveness of the input language. This may be indicated by the data types and expressions allowed in the program (e.g., Booleans, integers, arrays, pointers; linear arithmetic expressions, mixed logical and arithmetic expressions). We hope that powerful tools will be developed which can accurately analyze programs in more and more expressive languages.

References

1. R.S. Boyer, B. Elspas and K.N. Levitt, SELECT – A formal system for testing and debugging programs by symbolic execution, *Proc. of the Int. conf. on Reliable Software*, 234–245, 1975.
2. W.R. Bush, J.D. Pincus and D.J. Sielaff, A static analyzer for finding dynamic programming errors, *Software – Practice And Experience*, 30: 775–802, 2000.
3. P. Cousot *et al.* The ASTRÉE analyzer, *Proc. 14th European Symposium on Programming (ESOP 2005)*, LNCS 3444, Springer, 21–30, 2005.
4. D. Detlefs, G. Nelson, J.B. Saxe, Simplify: A theorem prover for program checking, *J. ACM*, 52(3): 365–473, 2005.
5. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe and R. Stata, Extended static checking for Java. *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 234–245, 2002.
6. C.A.R. Hoare, Assertions in modern software engineering practice, Keynote address, *26th Int'l Computer Software and Applications Conf. (COMPSAC)*, Oxford, England, Aug. 2002.
7. Tony Hoare, The verifying compiler: A grand challenge for computing research, *J. of the ACM*, 50(1): 63–69, 2003.
8. J.C. King, Symbolic execution and testing, *Comm. of the ACM*, 19(7): 385–394, 1976.
9. A.K. Mackworth, Constraint satisfaction, in: *Encyclopedia of Artificial Intelligence*, (ed.) S.C. Shapiro, Vol. 1, 205–211, John Wiley, New York, 1990.
10. J. Zhang, Symbolic execution of program paths involving pointer and structure variables, *Proc. of the 4th Int'l Conf. on Quality Software (QSIC)*, 87–92, 2004.
11. J. Zhang and X. 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.
12. J. Zhang, C. Xu and X. Wang, Path-oriented test data generation using symbolic execution and constraint solving techniques, *Proc. 2nd Int'l Conf. on Software Engineering and Formal Methods (SEFM)*, 242–250, 2004.
13. M. Zitser, R. Lippmann and T. 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.