Automatic Software Verification
Lecture 5
Symbolic vs. Concrete Testing

Lecture date:
21st April, 2015

Introduction

We begin by giving a few definitions.

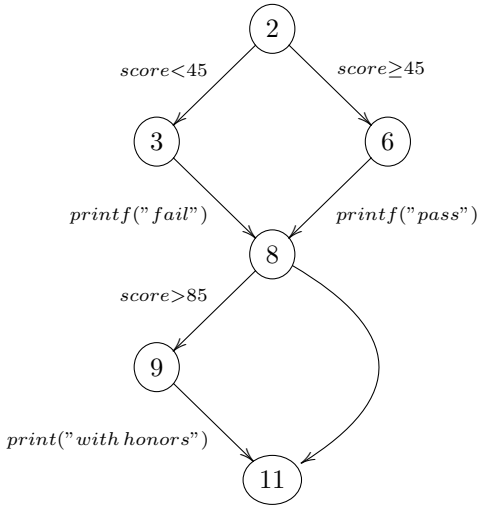
Definition 1. A *Program Path* is a path in the control flow of a program. It can start and end at any point. This definition is particularly appropriate for imperative programs.

Definition 2. A *Feasible* program path is reachable. In other words, there is some input that leads to the execution of that path.

Definition 3. An *Infeasible* program path is unreachable. There is no input that leads to the execution of that path. In some cases, this means some code is *Unreachable*.

```
1  void grade(int score) {
2          if (score < 45) {
3                  printf("fail");
4          }
5          else {
6                  printf("pass");
7          }
8          if (score > 85) {
9                  printf("with honors");
10         }
11         // ...
12 }
```

(a)



(b)

Figure 1: An Infeasible Program Path

Example 4. Figure 1 shows a program with an infeasible path. Specifically, the path $3 \to 8 \to 9$ is infeasible. There is no value for *score* that will cause the program to execute the body of the *else* clause, as well as enter the second condition.

Real programs have many infeasible paths. This makes concrete testing ineffective. Therefore, we turn to symbolic testing. Intuitively, we write the path as a formula, and ask the Theorem Prover whether the formula is satisfiable.

We will review three testing methods: 1) Fuzzing, 2) symbolic exploration, and 3) Concolic testing.

Random (Fuzzing) Testing

In this form of testing, program paths are executed on random and unexpected input. This input can be both valid and invalid. This form of testing supports white-box testing, where the internal structures and workings of an application are tested. It also supports black-box testing, where the functionality of the program is being tested, but its internal structures and workings are not. This method also supports grey-box testing, which is a combination of both white-box and black-box testing.

This testing technique has been found to be effective. It was used to test the reliability of Unix programs, as well as network protocols and system libraries. It is usually used by instrumentation, i.e. automatic processes which generate random data, and passes it to the program.

```
1  int f(int *p) {
2          if (p != NULL) {
3                  return q;
4          }
5          // ...
6  }
7
8  if (x == 10001) {
9          // ...
10         if (f(*y) == *z) {
11                 // ...
12         }
13 }
```

Figure 2: A snippet of a complex program

However, it is very difficult to scale this technique to complex programs which have many paths. See, for instance, the program in figure 2. The probability of randomly selecting $x = 10001$, and the correct values for y and z are very slim. Therefore, the paths going through line 11 will probably never be tested.

Symbolic Exploration and Testing

In this method, an interpreter follows the program. It assumes symbolic values as inputs, rather than concrete values. It tracks the state of the program, such as variable values, in a symbolic manner, and updates this state according to the executed instructions. This method does not follow a single program path, but rather a class of paths, defined by rules on the input which makes these paths feasible.

For each branch in the program, the interpreter attempts to prove, using a Theorem Prover, whether it is feasible or not. The Theorem Prover is given the symbolic state of the program. This method can therefore identify both unreachable code, feasibility of program paths that lead to program failure (such as crashes, or invalid calls with invalid inputs), and what class of inputs causes the program to crash.

One can generate a *Symbolic Execution Tree* characterizing the program paths followed during the symbolic exploration. Each node in the tree is associated with the execution of a single statement. It is labelled with the statement number, as well as the symbolic state of the program. Each transition between statements is a directed arc connecting the associated nodes.

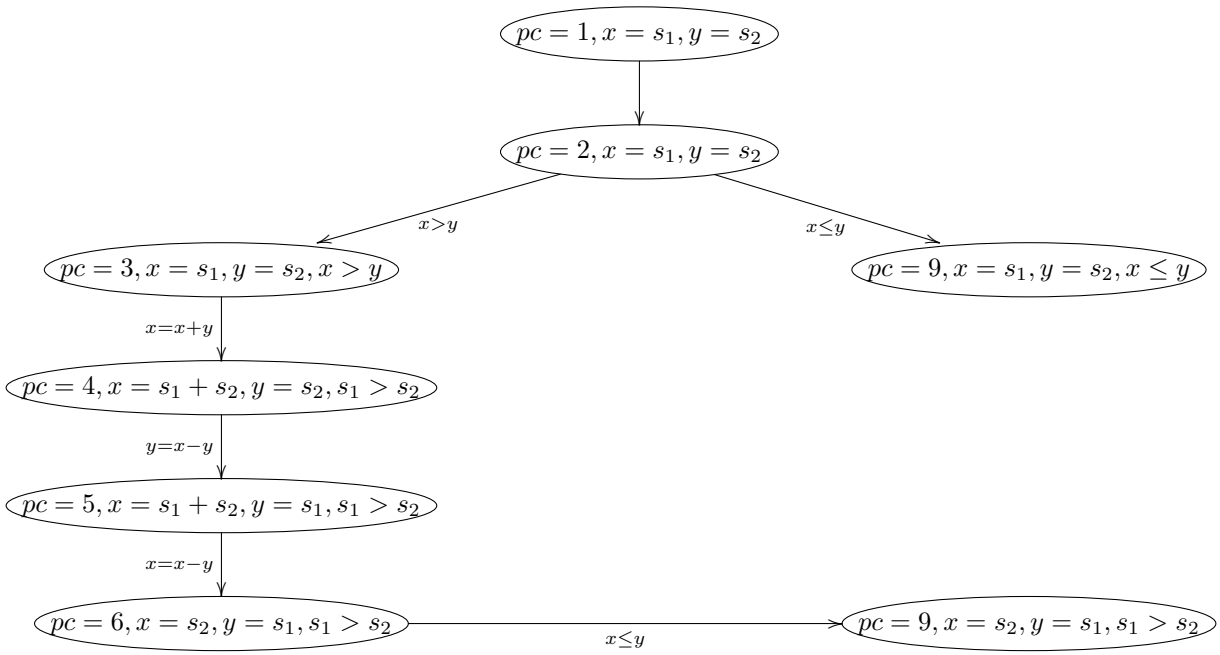
In case of branching, the associated node has two outgoing arcs, labelled with the true and false conditions of each possible outcome, leading to the nodes associated with the execution of the two branch parts.

The Symbolic Execution Tree is constructed during the symbolic evaluation of the program. The construction of each node requires writing down the state of the program. The construction of each edge requires calling the Theorem Prover, to discern the next node's state.

Examples

```
1  int x,y;
2  if (x > y) {
3          x = x + y;
4          y = x - y;
5          x = x - y;
6          if (x > y)
7                  assert(false);
8  }
9  // ...
```

(a) Code listing



(b) Symbolic execution tree

Figure 3: A simple example

Example 5. Figure 3 shows the code and symbolic execution tree of a simple program. The nodes show that the interpreter does not have the actual values for x, and y. Rather, it maintains symbolic values s_1, and s_2, respectively.

Once it reaches the branch where $pc = 2$, the interpreter asks the Theorem Prover which of the possibilities is *feasible*. Since the Theorem Prover has no prior knowledge on x, and y, both paths are feasible. Therefore, the interpreter needs to explore both paths. This method can be used to find if a code segment is *unreachable.* We will see that for the branch in $pc = 6$, only one branch part is feasible.

As the interpreter passes over states $pc = 3$ to $pc = 6$, it simulates the operations in a symbolic manner. It can be seen in state $pc = 4$ that $x = s_1 + s_2$, since these are the symbolic values for x, and y. The Theorem Prover also performs simplification, as can be seen in step $pc = 5$, where the state is that $y = s_1$, where without simplification it would be $y = s_1 + s_2 - s_2$.

When the interpreter reaches the state $pc = 6$, it encounters another branch. However, the Theorem Prover can now prove that $\neg(x > y)$. Therefore, only one path is feasible, specifically the arc to state $pc = 8$, where $x \leq y$. The path to state $pc = 7$ is infeasible, so the interpreter does not explore it. If this were not the case, the interpreter would have reported a possible bug, with the rules on the input which would generate it.

As part of the simplification process, the state $pc = 4$ changed the fact $x > y$ to $s_1 > s_2$. Since the value of x has changed, it is not necessarily true that $x > y$ at state $pc = 4$, only that $s_1 > s_2$. Note that $s_1 > s_2$ is already provable in state $pc = 3$, where $x = s_1, y = s_2, x > y \implies s_1 > s_2$.

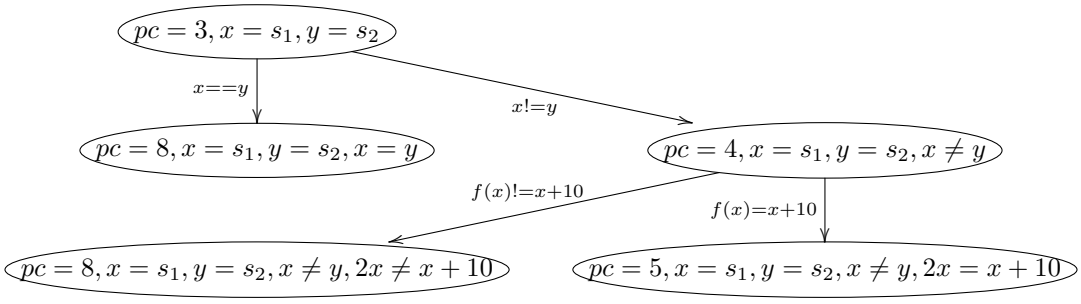
The decision of where to make such transitions can sometimes be problematic. There are many such facts that can be extracted, and having the interpreter memorize all of them may be inefficient. Not every fact is important, or used in the exploration, and storing it is wasteful. In general, the interpreter stores *Prime implicants*. A *prime implicant* is a known fact which cannot be implied by a more general known fact. Other facts can be implied when necessary.

The Theorem Prover has two main functions: 1) Detect which program paths are feasible and infeasible, and 2) learn facts about the program.

The construction of this interpreter is such that it does not report false positives. If it reports a bug, then this bug is real, and it can occur given a particular input. Without learning additional facts about the program and its execution, the interpreter will hold a more general program state. This means that it will not be able to show that some paths are infeasible, show bugs that cannot be reached, and return false positives. In this manner, this method is *sound*.

```
1  int f(int x) { return 2 * x; }
2  int h(int x, int y) {
3          if (x != y) {
4                  if (f(x) == x + 10) {
5                          abort(); /* error! */
6                  }
7          }
8          return 0;
9  }
```

(a) Code listing



(b) Symbolic Execution Tree

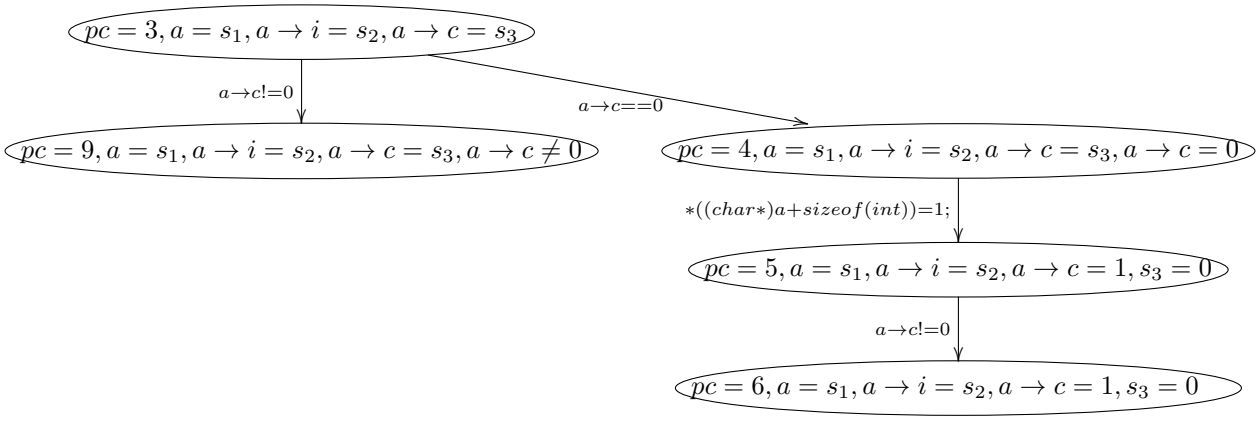
Figure 4: A simple example with an inlined function call

Example 6. Figure 4 shows a program path which calls a function. In this case the interpreter has inlined the function. Therefore, the Theorem Prover knows that $f(x) = 2x$.

The symbolic execution tree shows that there is a feasible program path leading to the bug on state $pc = 5$. It has also found the values leading to the bug, using only symbolic exploration. It can now find the weakest precondition leading to the error.

```
1  struct foo {int i; char c;}
2  bar(struct foo *a) {
3          if (a->c == 0) {
4                  *((char*)a + sizeof(int)) = 1;
5                  if (a->c != 0) {
6                          abort(); /* Error! */
7                  }
8          }
9          // ...
10 }
```

(a) Code listing



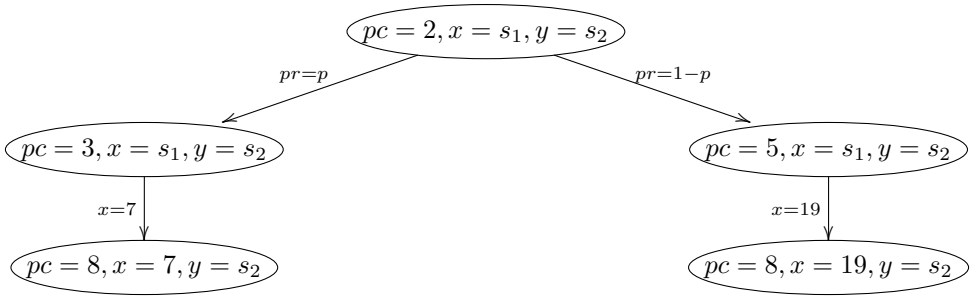
(b) Symbolic Execution Tree

Figure 5: A simple example with pointers

Example 7. Figure 5 shows a program which uses pointers. In this case, the interpreter maintains a symbolic value for a, the address of the structure. It also maintains symbolic values for $a \to i$, and $a \to c$, the elements within the structure. The symbolic execution path shows that the bug can be reached.

```
1  int x, y;
2  if (nondet()) {
3          x = 7;
4  }
5  else {
6          x = 19;
7  }
8  // ...
```

(a) Code listing



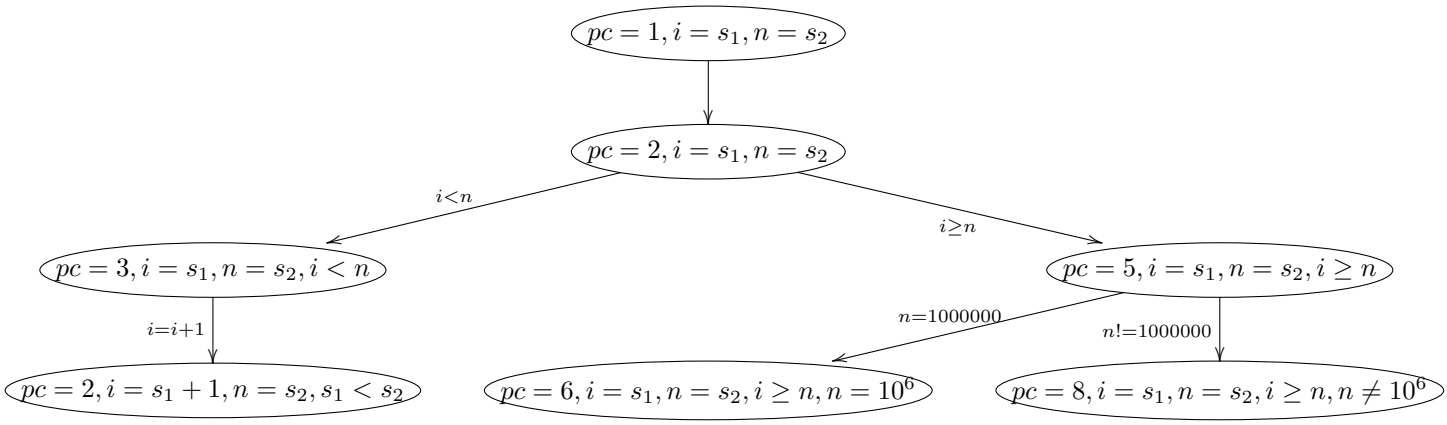
(b) Symbolic Execution Tree

Figure 6: A simple example with non-determinism

Example 8. Figure 6 shows a program which uses a function with a non-determinate outcome. The interpreter handles this by entering both program paths. This behaviour can arise when using e.g. library functions, or concurrency. If the behaviour is not real non-determinism, but modelled as such due to incomplete information, then infeasible flows may be undetected as such, and the interpreter may return false positive bugs.

```
1  int i;
2  while (i < n) {
3          i=i+1;
4  }
5  if (n == 1000000) {
6          abort(); /* error! */
7  }
8  // ...
```

(a) Code listing



(b) Symbolic Execution Tree

Figure 7: A simple example with a loop

Example 9. Figure 7 shows a program containing a loop. It can be seen that the interpreter will explore the loop's body again and, if necessary, many more times. If the interpreter were to explore using Depth First Search (DFS), there would be a danger for it to enter an infinite loop due to the loop in the program. However, the interpreter has found a feasible program path that causes the crash, without entering the loop at all.

Scaling Issues for Symbolic Exploration

There are still some challenges that make symbolic exploration infeasible for large and complex programs.

Limitations of Theorem Provers

Theorem Provers can handle linear problems well. For instance, integer linear arithmetic is done using Presburger arithmetic. However, non-linear problems, such as proving $x \cdot x \cdot x > 0$ or $x \cdot y \cdot z > 0$, are more problematic. For instance, finding the roots of a polynomial in discrete integers is hard.

```
1  if (x*x*x > 0) {
2          if (x > 0 && y == 10) {
3                  abort(); /* error */
4          }
5  }
6  else {
7          if (x > 0 && y == 20) {
8                  abort(); /* error */
9          }
10 }
11 // ...
```

Figure 8: Example program with non-linear arithmetic

Example 10. Figure 8 shows an example program with non-linear arithmetic. The Theorem Prover cannot prove that $x^3 > 0$ if and only if $x > 0$. Therefore, it cannot prove that program paths going through line 8 are infeasible. This means that it may show these paths as false positive bugs.

External Calls

In general, program code is not stand-alone. It calls, and is called from, external and library functions. To allow the Theorem Prover to learn facts about these functions, we may need to write an entire theory encompassing their behaviour.

```
1  FILE * fp;
2  fp = fopen("test.txt", "w");
3  if (fp) {
4          struct stat buffer;
5          if (stat("text.txt", &buffer) != 0) {
6                  abort(); /* error */
7          }
8          // ...
9  }
10 // ...
```

Figure 9: Example program with non-linear arithmetic

Example 11. Figure 9 shows an example program with calls to the standard library. We know that if *fopen* successfully opened a file for writing, it should exist, and therefore calling *stat* on that file should succeed. However, the Theorem Prover does not know that without a complete theory for the standard library. In this case, the interpreter will find a false positive bug on line 6.

Number of Calls to the Theorem Prover

Relative to direct execution, calling the Theorem Prover is slow. Calling it for every statement execution makes the entire process of symbolic exploration slow as well. Additionally, in case of loops with many cycles, what may take a direct execution approach several milliseconds, may take the symbolic exploration process an unacceptable amount of time.

```
1  int i=0;
2  while (i < n) {
3          i = i + 1;
4  }
5  if ((n == 1000000) && (i > n)) {
6          abort(); /* error */
7  }
8  // ...
```

Figure 10: Example program with a loop with many cycles

Example 12. For example, see figure 10. We have seen that the interpreter can handle the condition $n = 10^6$ even without entering the loop. However, to proving that $i = n$ or $i > n$, the interpreter has to iterate the entire loop, calling the Theorem Prover approximately one million times. Even at 1 millisecond a call, the interpreter will need more than 15 minutes to explore 7 lines of code. The alternative is not proving that $i = n$, and discovering a false positive bug on line 6.

Concolic Testing

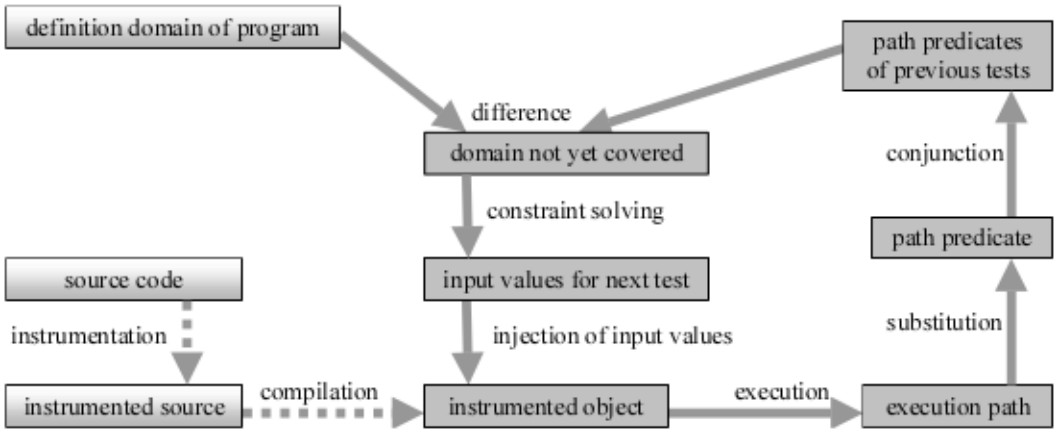


Figure 11: The Concolic Testing Algorithm (*Williams, Nicky; Bruno Marre; Patricia Mouy (2004). "On-the-Fly Generation of K-Path Tests for C Functions"*)

Concolic Testing attempts to reconcile between Fuzzing and Symbolic Exploration. It uses concrete, random testing to test a program path. Additionally, every time a single path from a branch is selected, the information is sent to the Theorem Prover to find rules or values for the input variables, so that the branch will be selected on the next iteration of input variables. In essence, the Theorem Prover is called for every branch, but not for every executed statement. Figure 11 shows a flow diagram of the Concolic testing algorithm.

This method forgoes storing the state in a pure symbolic fashion. This means it may miss some program paths, and be unable to prove some paths are feasible. However, this method still does not report false positive bugs. It is still *sound*.

```
1  int double(int v) { return 2*v; }
2  void testme(int x, int y) {
3          int z = double(y);
4          if (z == x) {
5                  if (x > y + 10) {
6                          abort(); /* error */
7                  }
8          }
9          // ...
10 }
```

Figure 12: A simple program

Example 13. Figure 12 shows a simple program. We will show how the Concolic interpreter passes over the program and finds input leading to the program error.

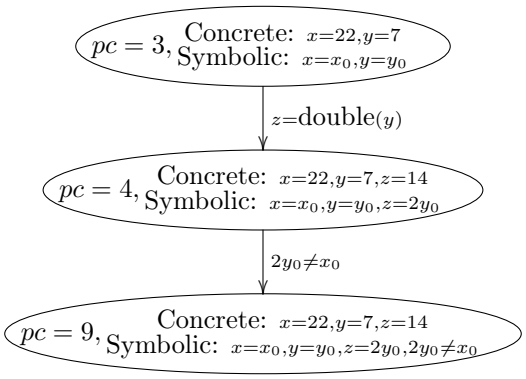


Figure 13: Execution path for $x = 22$, and $y = 7$.

The first step is selecting random input values, say $x = 22$, and $y = 7$. Figure 13 shows the execution path for these input values. The interpreter passed over a single branch statement, with the condition $2y_0 \neq x_0$. Therefore, it will try to select random input values that negate this condition, so as to enter the alternative path.

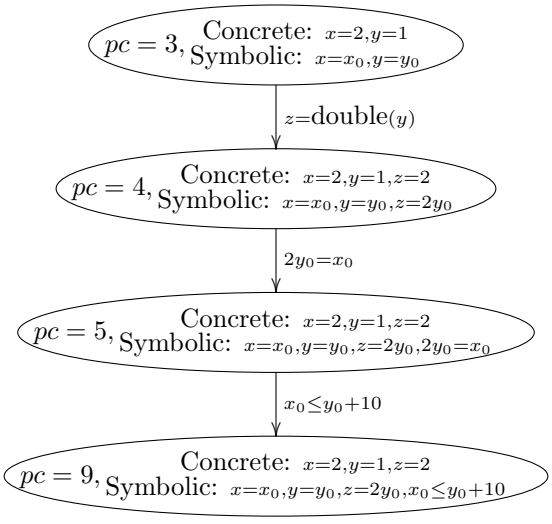


Figure 14: Alternative execution path for $x = 2$, and $y = 1$

Figure 14 shows the execution path that occurs when $x = 2$, and $y = 1$. There are now two branch statements.

For the first statement we have already explored the alternative branch. The condition for the second statement is $x_0 \leq y_0 + 10$. Therefore, the interpreter will try to select random input values that negates this condition, i.e. solve for $(2y_0 = x_0) \wedge (x_0 > y_0 + 10)$.

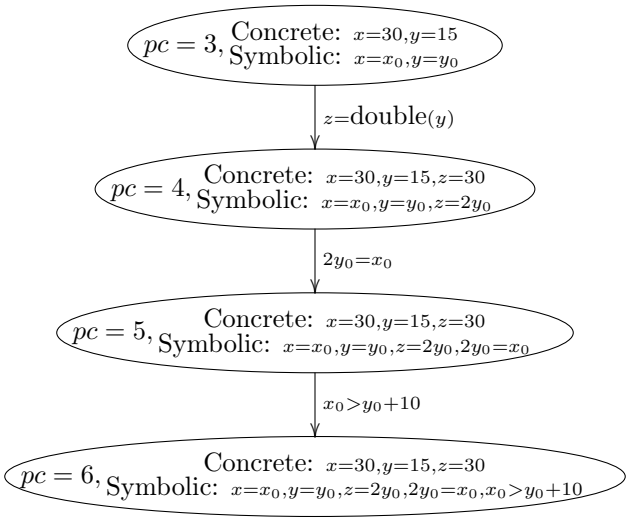


Figure 15: Another alternative execution path, where $x = 30$, and $y = 15$. This execution path ends with an error.

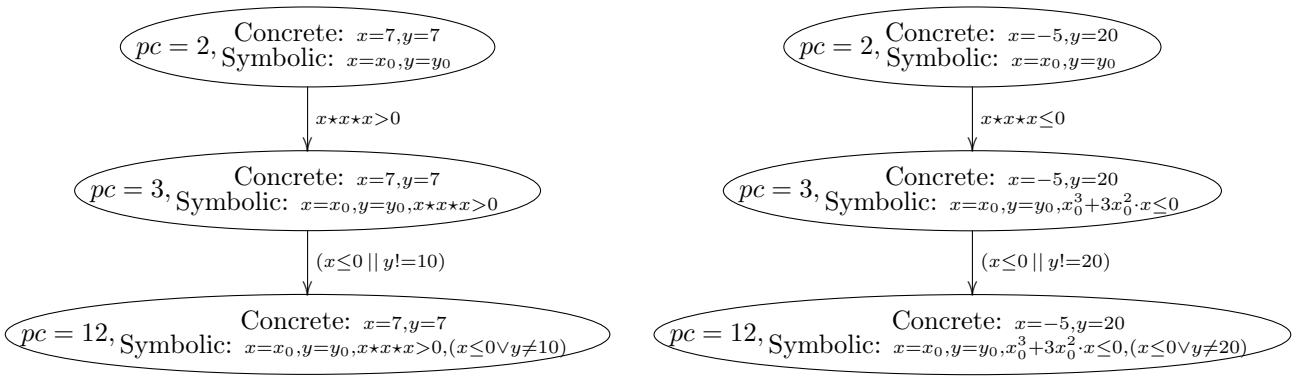
Figure 15 shows a third execution path, where $x = 30$, and $y = 15$. In this execution path we see that line 6 is reached, and there are possible input values that lead to an error. Specifically, those values are $x = 30$, and $y = 15$.

In general, the Concolic testing methods tests the program code using symbolic values where the Theorem Prover will do so efficiently (e.g. linear arithmetic), and concretely in other cases. We will see this behaviour in the following examples.

```
1  foobar(int x, int y) {
2          if (x*x*x > 0) {
3                  if (x > 0 && y == 10) {
4                          abort(); /* error */
5                  }
6          }
7          else {
8                  if (x > 0 && y == 20) {
9                          abort(); /* error */
10                 }
11         }
12         // ...
13 }
```

Figure 16: Example program with non-linear arithmetic

Example 14. We review again the code segment in example 10, in figure 16. To overcome the challenge of non-linear arithmetic, Concolic testing uses linear underestimation. Therefore, we can expect to see to execution paths, when x is negative, and when x is positive.



(a) Execution path when x is positive

(b) Execution path when x is negative

Figure 17: Execution path for the program in figure 16

For instance, we assume the linear estimation for $f(x) = x^3$ is $f'(x) = x_0^3 + 3x_0^2 \cdot x$ (using the Taylor series), where x_0 is the first random choice for x. If x_0 is positive, we expect the next value for x to hold $x < \frac{x_0}{3}$, which will eventually be negative. On the other hand, if x_0 is negative, we expect the next value for x to hold $x > \frac{x_0}{3}$, which will eventually be positive. Both choices can be seen in figures 17a and 17b, respectively. It can be seen that the interpreter will never report an error on line 9, which would be a false positive. In these examples, we have not shown the exploration where $y = 10$.

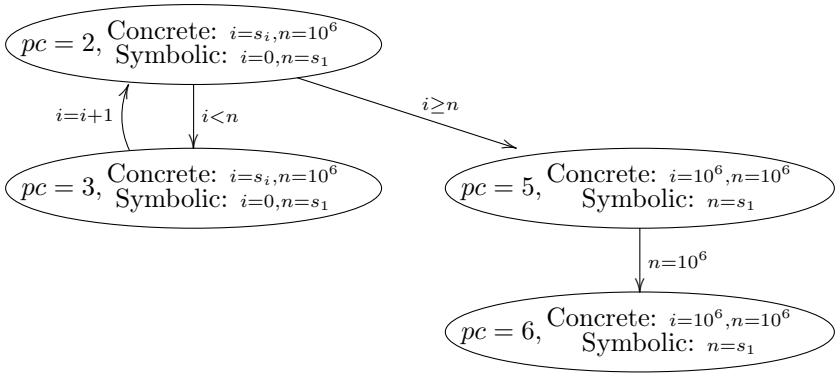
```
1  struct foo {int i; char c;}
2  bar(struct foo *a) {
3          if (a->c == 0) {
4                  *((char*)a + sizeof(int)) = 1;
5                  if (a->c != 0) {
6                          abort(); /* Error! */
7                  }
8          }
9          // ...
10 }
```

Figure 18: A simple example with pointers

Example 15. We review again the code segment in example 7, in figure 18. In this case, the interpreter will select only two values for a, since it is a pointer: 1) NULL, and 2) not NULL. The rest will be done in the same manner as in example 7, except that an error will be detected when $a = NULL$.

```
1  int i = 0;
2  while (i < n) {
3          i=i+1;
4  }
5  if (n == 1000000) {
6          abort(); /* error! */
7  }
8  // ...
```

(a) The code segment



(b) Execution path on the code above. s_i is a place-holder for the real value of i in every iteration of the loop.

Figure 19: A simple example with a loop

Example 16. We review again the code segment in example 9, in figure 19a. In this case, the interpreter will execute the entire loop in a concrete fashion. This means that the Theorem Prover will not be called. This behaviour is acceptable, since interpreting a loop a million times is much faster than calling the Theorem Prover a million times. It is also acceptable if the loop becomes infinite, since the interpreter will never return in this case, and an error will still be registered. Figure 19b shows a Concolic execution path on this code segment. The edges in the loop iterations between states $pc = 2$, and $pc = 3$, do not involve calling the Theorem Prover. The Theorem Prover is called for the transition between states $pc = 5$, and $pc = 6$.

```
1  FILE * fp;
2  fp = fopen("test.txt", "w");
3  if (fp) {
4          struct stat buffer;
5          if (stat("text.txt", &buffer) != 0) {
6                  abort(); /* error */
7          }
8          // ...
9  }
10 // ...
```

Figure 20: Example program with non-linear arithmetic

Example 17. We review again the code segment in example 11, in figure 20. In this case, external library functions are called in a concrete manner only. If they work well with the input variables and environment on the testing system, then no bugs are discovered. This means that some bugs may be missed. However, it still maintains that there are no false positive bugs reported.

Tools

We introduce two tools that can be used for Concolic testing: 1) DART and 2) SAGE.

DART

From the paper introducing DART (Godefroid, Klarlund, Sen, *DART: Directed Automated Random Testing*):

> We have implemented DART for programs written in the C programming language. Preliminary experiments to unit test several examples of C programs are very encouraging. For instance, DART was able to find automatically attacks in various C implementations of a well-known flawed security protocol (Needham-Schroeder's). Also, DART found hundreds of ways to crash 65% of the about 600 externally visible functions provided in the oSIP library, an open-source implementation of the SIP protocol.

SAGE

From the paper introducing SAGE (Godefroid, Levin, Molnar, *SAGE: Whitebox fuzzing for security Testing*):

> Since 2007, SAGE has discovered many security-related bugs in many large Microsoft applications, including image processors, media players, file decoders, and document parsers. Notably, SAGE found approximately one-third of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7. Because SAGE is typically run last, those bugs were missed by everything else, including static program analysis and blackbox fuzzing. Finding all these bugs has saved Microsoft millions of dollars as well as saved world time and energy, by avoiding expensive security patches to more than one billion PCs. The software running on your PC has been affected by SAGE.
>
> Since 2008, SAGE has been running 24/7 on approximately 100-plus machines/cores automatically fuzzing hundreds of applications in Microsoft security testing labs. This is more than 300 machine-years and the largest computational usage ever for any Satisfiability Modulo Theories (SMT) solver, with more than one billion constraints processed to date.
>
> SAGE is so effective at finding bugs that, for the first time, we faced "bug triage" issues with dynamic test generation. We believe this effectiveness comes from being able to fuzz large applications (not just small units as previously done with dynamic test generation), which in turn allows us to find bugs resulting from problems across multiple components. SAGE is also easy to deploy, thanks to x86 binary analysis, and it is fully automatic. SAGE is now used daily in various groups at Microsoft.

Conclusion

Concolic testing is a very powerful method. However, it still has many issues, including scaling and performance. Future progress in symbolic reasoning can help both symbolic exploration, as well as Concolic testing.