Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2021

Martin Steffen

Contents

Chapter 7 Symbolic execution

Learning Targets of this Chapter

The chapter gives an not too deep introduction to *symbolic* execution and *concolic* execution.

Contents

What is it about?

7.1 Introduction

The material here is partly based on [2] (in particular the DART part). The slides take inspiration also from a presentation of Marco Probst, University Freiburg, see the link here, in particular, some of the graphs are reused and adapted from that presentation. More material may be found in the survey paper [1].

Symbolic execution is a quite "old" technique, one or the starting point for it is [3] from 1976. It's a technique natural also in the context of *testing* (and in the chapter, we talk also about some aspects of testing). We cover also the ideas behind *concolic* execution, a portmanteau word meaning "concrete and symbolic". Symbolic execution is also used in compilers, for optimization and code generation.

```
f(int x, int y){
  if (x*x*x* > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail ();
    }
  }

  complete();
}
```

Listing 7.1: Sample code

Let's take a look at Listing 7.1. The code has no particular purpose, except that it will be used to discuss testing, symbolic execution, and also concolic execution. The function has two possible outcomes, namely success or failure, represented by calls to corresponding procedures. Note that non-termination is not an issue, there is no loop in the procedure. In general, symbolic execution works best or most straightforward on straight-line programs and loops pose challenges for symbolic execution. The problems with loops are similar the challenges they pose for bounded model checking . Indeed, BMC shares some commonalities with symbolic execution: both are making use of SAT/SMT solving.

How to analyse a (simple) program like that?

One has different options there; and can of course pursue more than one of them. One standard thing to do is **testing**. Testing is probably *the* most used method for ensuring software (and system) “quality”.

Testing is a broad field and has very many aspects, and there are very different approaches to testing and techniques and different testing goals. Those techniques are also used in combination and in different phases of software engineering cycle.

For function bodies in isolation like the ones shown, *unit testing* is a way to go (perhaps a part of a larger testing set-up for the whole product).

One could do “verification”, whatever that means. The term could include code review, or a formal verification of the code towards a specification, perhaps with the help of a theorem prover. Later we will disucss symbolic and concolic execution. Before we do that, what about model-checking?

Model-checking a program like that is challenging. Model-checking methods and corresponding (temporal) logics are mostly geared towards concurrent and reactive programs anyway. In particular, standard model checking techniques are not very suitable for programs involving data calculations. The given code is a procedure with *input* and its behavior is *determined* by the input. So, *given* the input, it’s a deterministic (and sequential) problem and with a concretely fixed input, there is also no “state-space explosion”. Generally, though, the problem is *infinite* in size, if one assumes the mathematical integers as input, resp., unmanagably huge, if one assumes a concrete machine-representation of integers, i.e., for practical purposes, the state space is basically infinite, even though the program is tiny.

Of course, common sense would tell that if the program would works for having $x = 2345$ and $y = 6789$, there is no reason to suspect it would fail for $x = 2346$ and y unchanged, for example. In that particular tiny example, that is clear from the fact that those particular numbers are never even mentioned in the code, they are nowhere near any corner case where one would expect trouble.

This way of thinking (what are corner cases) is typical for testing, and is obvious also for unexperienced programmers (or testers). Of course it is based on the assumption that the code is available, as the intuitive notion of “corner case” rests on the assumption one can analyze the code and that one sees in particular which conditionals are used. For instance, there’s no way of knowing which corner cases the `complete()` might have, should it have access to those variables `x` and `y`, except perhaps some “usual suspects” like uninitialized value, 0, `MAXINT` and +/- 1 of those perhaps.

There are many forms of testing, in general, with different goals, under different assumptions, and different artifacts being tested. The form of (software) testing where the code is available is sometimes called *white-box testing* or *structural testing* (the terms white-box and black-box testing are considered out-dated by some, but widely used anyway).

Coverage

The intuitive thinking about "corner cases" basically is motivated by making sure that all possible "ways" of executing the code or actually done. In testing that's connected to the notion of *coverage*. In the context of white-box testing, one want to cover "all the code". What that exactly means depends on the chosen coverage criterion or criteria. The crudest one (which therefore is not really used) would be *line coverage* that every line must be executed and covered by a test case. It's not a useful concept: it would allow the tester to claim 100% line coverage if the program would be formatted in a single line... That's of course silly, so typically, criteria are based on covering elements of the programs represented by a *control-flow graph* (CFG, see the pictures later), and then one speaks about node coverage, or edge coverage, or further refinements, depending on the set-up. For instance, if one had a language that supports composed boolean conditions, and if one had a CFG representation that puts such composite conditions into *one node* of the CFG, then covering only that node, or covering both true and false branch of that node will not test all the individual *contributions* of the parts of the formular to that true-or-false condition. If want wants more ambitious coverage criteria, one may that those into account as well, which would be better than simple edge coverage.

So, there are very many coverage criteria. Known ones include

- node coverage
- edges coverage, condition coverage
- combinations thereof, and
- path coverage

They are defined to answer the question

When have I tested "enough"?

Agreeing on some coverage criterion then measuring how much coverage a test gives is one thing. Another important and more complex thing is to figure out what test cases are needed to achieve good coverage, and then arrange for that automatically. In the given example, that may be simple. The example is tiny, one can see a few boolean conditions and easily figure out inputs that cover each decision as being both true (for one test case) and false (for another). Practically, one may choose the exact corner-cases and then one off, since one should not forget that the *real* goal is not "coverage", the real goes is to make sure that a piece of code has no errors, or rather more realistically: testing should have a better than random chance to detect errors, should there be some. As a matter of fact, one common source of errors is getting the corner cases wrong (like writing $<$ in a conditional instead of $\leq$ or the other way around, especially in loops), which is sometimes called off-by-one error. So, if the code contains a simple, non-compound condition $x > 0$, choosing as input $x = 700$ and $x = -700$ may cover both cases (= 100% edge coverage for that conditional), but practically, choosing $x = 1$ and $x = 0$ may be better.

But anyway, to achieve good "coverage" and/or good testing of corner cases, the **real question** is:

> How to do that *systematically* and *automatically*? How to generate necessary input for the test-cases to achieve or approximate the chosen coverage criteria?

That in a way a the starting point of *symbolic execution*, which has its origin in testing. As coverage, it's based typically on something more ambitious than edge coverage or some of the refinements of that. It's based on **path coverage**. Path coverage requires that each *path* from the beginning of the procedure till the end is covered. If there are loops, there are infinitely many paths, which explains the mentioned fact, that loops are problematic. The method is called "symbolic" as it's not about *concrete* values to cover all paths (if possible). So, if one has a condition $x > 0$ as before, it's not about choosing $x = 700$ and $x = -700$ (or maybe better $x = 1$ and $x = 0$). Symbolically, one has two situations: simply $x > 0$ and it's negation $\neg(x > 0)$ (which corresponds to $x \leq 0$), i.e., the two possible outcomes of a condition with that conditions corresponds to two *constraints*. It should be noted: even if, in the presence of loops, there are infinitely many paths 100% path coverage does not cover *all reachable states*, as different values can lead to the same path. That means full path coverage is not the same as full verification or model checking.

Programs typically contain more control structure than just one or two conditions. So, symbolic execution just takes *all paths*, each path involves taking a number of decisions along its way, every one either positively or negatively, and collects all constraints in a big conjuction.

There is more to say about symbolic execution as a field, but that's one core idea in a nutshell.

Path coverage is often considered as too ambitious as coverage criterion. Of course, sometimes tests cannot cover 100% of the simpler critera as well. Nodes that belong to dead code cannot be covered. In a unit with dead code, one cannot achieve 100% coverage. But perhaps one should, since indirectly, dead code may be a sign of a problem as well (only one cannot test dead code in a conventional way, and in a way, there may be no point to test it either). In the presence of loops, there are typically *infinitely many paths*. That means, no matter how many test cases one comes up with, the coverage is always 0%, so in this plain form, one cannot use path coverage to measure if one has tested "enough". Note also: the fact that there are infinitely many paths is not the same as saying that the program itself is non-terminating (for some input). The notion of paths (in the context of path coverage) refer to paths through the control flow graph (CFG), which is an abstraction. The paths may or may not correspond to paths through the graph done when *executing* the actual program. That also means, there may be paths in the CFG that are unrealizable, and in particular, all loops in the progam may actually terminate, but that's something one cannot see in the CFG, where one can see just a cycle in the graph.

Let's revisit the small progrom from earlier, from Listing 7.1. Figure 7.1 shows the corresponding control-flow graph. The graphical "design" used in that figure is sometimes called flow-graph, using some conventions. For instance that the condinals are represented by diamond- or rhombus-shaped nodes, the input-nodes by rhomboid etc. For us, those conventions don't matter much (and they may also vary from presentation to presentation). But maybe they help to visualize the notion of control flow graph. Indeed, control-flow graphs are not primarily a visialization, the are often concrete data structures inside a compiler or model checker, and important intermediate representation, serving various analysis, optimization, and code generation purposes.

Here, coverage is defined in terms of paths through the control-flow graph, and thus also (an implementation of) symbolic execution is based on some form of control-flow graph.

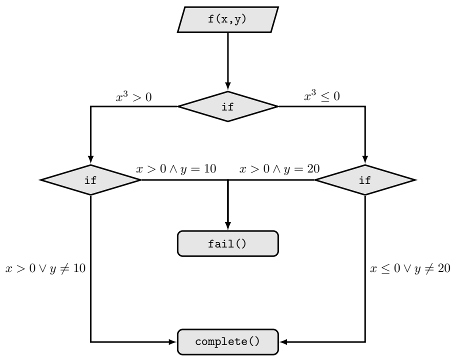
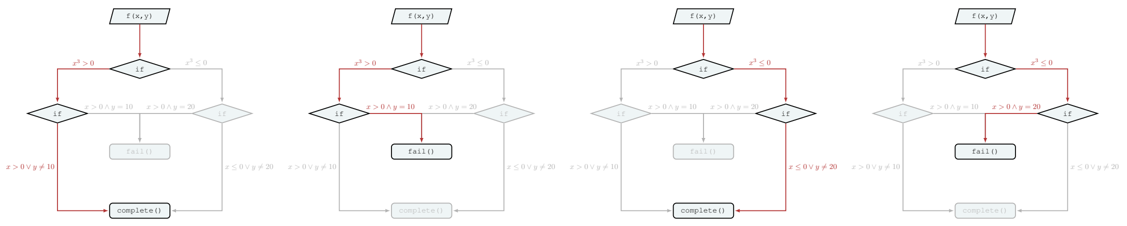


Figure 7.1: Control-flow graph/flow chart

The boolean conditions on the edges correspond to the the condition in case the if- resp. the else-case is taken of the corresponding if-statement. The two branches are mutually explusive (in conventional, deterministic programs), the false-branch is the negation of the true-branch. It should be noted that we assume that there the conditions are are side-effect free. That's generally good programming style, even in case the programming language would support it. Besides, it's of course not real restriction. It's easy to transform "dirty" programs with side-effects in conditions into one that is more disciplined that way. An actually, should the programmer turn a deaf ear on advice like "boolean conditions are better side-effect free", the compiler (or model-checker or analysis tool) will take care of it. Normally, control-flow graphs are not meant for human consumption (unless one uses a graphical programming notation, UML etc), it's an internal representation of the tool, for the purpose of analysis. The flow graph may not even be to represent the control-flow in source code syntax, but perhaps for a lower level intermediate code representation. Keeping that clean helps with whatever one intends to use the CFG for, like code generation, analysis, optimization, etc. Or in our case, symbolic execution, which is a form of analysis anyway. Thus, we it's perfectly fine and realistic to assume the conditions are side-effect free.

The control-flow graph of the program is very simple and there are only 4 different paths from the initial node to one of the terminal nodes. Those four path are shown in Figure 7.2.

Following a path accumulates the conditions as they appear on the positive, resp. the negative edge on the decisions being taken, depending on which decision is assumed the particular path take. If one follows a path from the beginning to the end, one has a boolean



(a) Path 1 (b) Path 2 (c) Path 3 (d) Path 4

Figure 7.2: Four different paths through the control-flow graph

constraint which consists of the *conjunction* of all the indidual boolean constraints. One such constraint is also called the **path constraint** or **path condition** for that particular path.

For instance, the first path, as marked in Figure 7.2a, has the path condition

$$(x^3 > 0) \land (x > 0 \land y \neq 10) \tag{7.1}$$

and the last path from Figure 7.2d, has the path condition.

$$(x^3 \leq 0) \land (x > 0 \land y = 20)\ . \tag{7.2}$$

Obviously, the condition for the first path can be simplified, plausibly to $x > 0 \land y \neq 10$. The one from equation (7.2) has no solution (in the assumed conventional interpretation on "numbers"). It corrsponds to the constraint "false". As for terminology: the corresponding path is **unrealizable.**

Figure 7.3 contains all three realizable paths, marked in red. It's not the same as dead code, but of course one cannot find input that covers that path, as the path condition is contradictory.

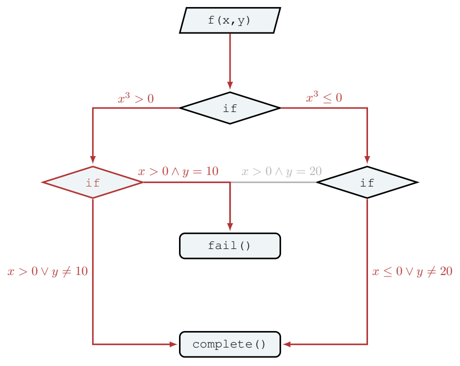


Figure 7.3: All realizable paths

Now the goal is: find a set of inputs to run program so that all *realizable* paths are covered, resp. find a method that automatically does so.

Random testing

Let's start with perhaps the most "naive" way of testing, namely to run the program repeatedly with randomly generated inputs. "Naive" is perhaps an exaggeration, random testing has and is being used, (and studied evaluated and compared to other forms, it has been refined etc.) So it has its place, at least including randomized aspects into testing. We ignore in the discussion here is that in testing one needs among other things, also a way to judge the outcome of the tests. I.e., one needs a specification of the expected outcomes, or at least of the expectation what should or should not happen (like that one does not want to see certain general errors). In the small program from Listing 7.1 that's assumed given by the two possible termination outcomes, either properly completed or failed.

For us important is that testing, randomized or otherwise works with **concrete** input values, and does a **dynamic** execution of programs, observing its *actual* behavior and compare it against *expected behavior*.

Example 7.1.1 (Random testing). Let's use the program from Listing 7.1 resp. its control-flow graph for illustration. Testing means providing different inputs that lead to different outcomes, following potentially different paths, and let's assume the input is generated randomly. For instance, one could use $(x, y) = (700, 500)$, and $(x, y) = (-700, 500)$ ... The first pair of values results in the path of Figure 7.2a, the second pair in the path of Figure 7.2c, both ending in the non-erroneous end-state, i.e., the two test cases are passed.

The path from Figure 7.2d is unrealizable. But with the two inputs so far, the realizable from Figure 7.2b has so far been missed.

We shouldn't count the unrealizable 4th path one among the ones we missed. But the realizable path shown should be covered. In particular, it's one that would point to an error in the program, the other two so far found no bug.

The problem with this is: to randomly hit that particular path has an astronomically **low probability** (hitting $y = 10$ by chance is very unlikely, indeed). Actually, this way of testing, at least the way of selecting input, may even not even be called *white box*, as it ignores information inside the body of the function, for instance that $y = 10$ seem a profitable corner case. □

In defense of random testing one may say: it may be easy in this particular case, to pick more reasonable or promising input like $y = 10$. That's not just because the program is small. Note in particular, that x and y are also *not updated in* fancy ways (maybe conditionally updated, maybe even using pointers and other complications). One may have to invest heavily in complex theories that may be time-consuming to run before one can get a decent grip on improving on the randomness of the input. And, in a way, *symbolic execution* is an investment in theory (SMT solving) to find an alternative way of testing, thereby also going from a black-box approach for selecting the inputs to a white-box view.

To avoid a mis-conception: random testing is not synonymous with white-box testing. If one does random input testing the way described, and then used path coverage to measure how good the test suites have been, that's *white-box* testing: to *rate* the path coverage, one needs access to the code. It's only that the available white-box information is not taken into account for shaping the test cases in a meaningful way (except for perhaps stop testing, when one feels the random input has achieved sufficient node/edge/path/whatever-coverage).

So, how to get to the missing path from Figure 7.2b? One input that would do the job is, for instance $(x, y) = (145, 10)$, but hitting the concrete value $y = 10$ by chance, as said, as a *very* low probability.

But that's where working with *symbolic representations* can do better, where it's not about individual, concrete values, but sets of values, resp. symbolic or formulaic representations of sets of values. In symbolic executions, one works with path constraints or path conditions. The path condition corresponding to the so far missing path from Figure 7.2b, after simplification, is

$$x > 0 \wedge y = 10 \ .$$

7.2 Symbolic execution

We have essentially introduced the core idea of symbolic execution in the previous section, focusing on path constraints.

Perhaps it's worth iterating that, like in BMC, it's about *SMT-solving* (not just SAT solving), **sat-solving modulo theories**. That's about

boolean combinations of constraints over specific domains with specific *theories*

The theory or theories allows to express properties of values like integers or arrays, etc., that corresponds to data types used in the programming language used for the programs we are analyzing.

One should be aware, that theories may easily lead to undecidability of constraint solving. Integers with addition only have a decidable theory.[1] Add multiplication, and decidability of the theory goes out the window.

Undecidability is a real issue: how many programs use only integers and addition? One could claim that the programs mostly never use real mathematic integers, but just a finite portion of them (up-to `MAX-INT`) so one is dealing with a finite memory, so that makes properties decidable. That's correct, and when dealing with integers and actual programs, one can make the argument, one should deal with the machine integers anyway to make it more realististic. Indeed, one can work with a theory capturing those "realistic" integers or "IEEE floating points", etc. But all those theories are non-trivial. So even if technically decidable (by being finite), it may be computationally too expensive to wait for an answer when doing SMT solving. And there are more data types than just numbers: there are dynamic data structures (linked lists, trees, etc.), and they are conceptually unbounded, as well. Again, one may posit that, in the real world, there is always some upper bound (`out-of-heap-space`, `stack-overflow`), but it's unrealistic to capture

[1]This specific theory is known as Presburger arithmetic.

those limitations in a decidable theory and hope the constraint solver will handle it thereby. It would even make no sense conceptually, if one is doing "unit testing": the procedure under test may or may not have out-of-memory problems depending on factors *outside* the unit. For instance on how much heap space is already taken away by other data structure in the program.

Anyway, one has to face the sad fact that one will encounter constraints that are either formally undecidable or untractable; in some way, there's not much practical difference either way. In some not too far-fetched situations, constraint solving may simply not work.

We come back to that later: **concolic execution** is an extension of symbolic execution that addresses exactly that problem: what can I do if my constraint problem exceeds the capabilities of the used SMT solver. But first we finish up with symbolic execution by looking at a super-simple example, but without adding much new technical content to the material, it's more like rubbing it in a bit more. One difference to the previous example, though, is that now a variable involved in the program is *assigned* to, i.e., changes its value on the path(s) through the execution

Let's have a look at the code from Listing 7.2 resp. the CFG from Figure 7.4

```
1  y = read();
2  y = 2 * y;
3
4  if (y==12) {
5     fail();
6   }
7
8  complete();
```

Listing 7.2: Sample code

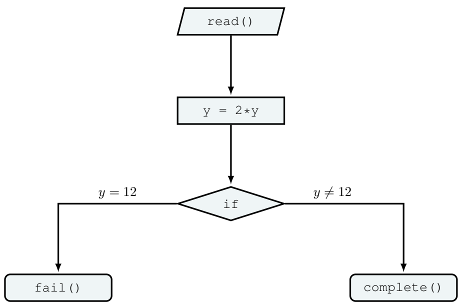


Figure 7.4: CFG

In the C-style code, the "equation" sign of course represents assignments, and the == is a comparison. In the constraints and the annotation on the edges, we use = for comparison, and in the text here, for clarity, we use `:=` for assignments. The difference between assignments and equations should be clear. If not, looking at line 2 of the code snippet: `y := 2 * y` is definitely not the same as the equation $y = 2y$). The latter is unsatisfiable using standard numerical theories.

Let's also introduce the variable s for containing the result of the `read()`-operation. The code contains two asignments, `y := read()` and `y := 2*y`. That leads to two constraints,

$$y = s \quad \text{and} \quad y = 2s$$

at the corresponding points in the program. The branching point in line 4, leads to the two conditions $2s = 12$ and $2s \neq 12$.

As a side remark: we came to the constraint like $y = 2s$ that holds after line 2 by looking at the very simple example. Nore systematic would be to work with different instances or incarnation of the variables. Here with different versions of y, as this is the only variable being asigned to. Actually there are two versions of y, say y_0 after the first assignment and y_1 after the second. I.e., the constraint solver would have to deal with the two constraints

$$y_0 = s \qquad y_1 = 2y_0 \tag{7.3}$$

which is equivalent as far the purpose of the symbolic execution is concerned.

We have seen the same treatment of using different "versions" to represent mutable program variables also in the context of symbolic model checking (where we used unprimed and primed version to capture the pre- and the post-situation in representing the successor states (for the representation of the *pre*- or *post*-sets). And the "versioning" treatment was analogously done for bounded model checking, which also needed to capture paths.

Back to the program. To find, for instance, the erroneous outcome, the onstraint solver needs to solve the path constraint $2s = 12$ (or the slighly longer one from equation (7.3).

That's (in this case) child's play: the solution is $s = 6$.

However, the constraint containts multiplication. We shortly mentioned it before: the theory of natural numbers with addition and multiplication is undecidable.

In this particular example, the constraint is trivially solved by humans, and would pose not problem for constraint solvers. Indeed, the constraint $2s = 12$ is covered by a decidable theory, namely a restriction of the general case of addition and multiplication, where multiplication is restricted to involve only one variable multiplied with constants (so constraints like $xy > 0$ and also $x \times x = 23$ would violate that restriction).

A constraint like $2x + 17y < z$, using an inequation instead of equality, would still be ok: there are 2 variables but they are not multiplied *with each other*. Such restricted forms can be covered by *linear arithmetic*, which has a decidable theory. It's an important class of constraints. For strange historical reason, the field dealing with such (in)equations (and generalizing the question of satisfiability to the question of finding an *optimal solution*) is called *linear programming*. It's also know under the less strange name of *linear optimization*.

Here is a short (intermediate) summary of what's been said, symbolic execution for dummies. It works like this: take the code (resp. the CFG of the code), collect all paths into **path conditions**. A path condition is a big conjunctions of all conditions along each the path. Each single condition b will have one positive mention b in one continuation of the path, one negated mention $\neg b$ in the other continuation.

Then feed the contraints to an appropriate SMT constraint solver, in particular solve the constraints for paths leading to errors. The whole approach works best for loop-free programs (and we will not cover what could be done for loops).

Even if one leaves out loops, which are problematic and focuses on straight-line code, the path constraint themselves may not easily solvable.

Looking again at the code from Listing 7.1 and the paths through the CFG, the path constraints mention x^3 as part of their path constraints. In particular also for the realizable path from Figure 7.2b and the unrealizable one from Figure 7.2d.

With the numerical constraints non-linear, we are definitely leaving the safe ground of decidable theories. Many contstraint solvers would throw the towel when facing those, for instance by only accept linear constraints in numerical domains, or under other restriction. Most solvers would not be ready to deal with random math constraints.

What can one do?

What can one do, beyond throwing the towel and accept that SE won't cover all paths or won't work on many programs? Later we will cover so-called concolic testing, but that is only one possible way to address the limitations of constraint solvers.

First make some remarks on *other* ways, as well, even if we don't cover then. One thing one could do is involve humans in some way, in the spirit of theorem proving. Theorem provers typically can do a lot more than guiding a human through manual proof activities. There is a good deal of automation under the hood, including constraint solving and verification in many domains. And even if undecidable, one could give it a shot, maybe relying on heuristics that in practice can handle many situations. But still, any method that involves human assistance in logical argumentation in formal theories is probably hard to sell in most areas and unappealing for large programs. For most areas a technique is either automatic, or unused ...

One can also give up on the goal of full path coverage. Most testing approaches don't do try that anyway. Random testing that we touched upon makes not attempt in the direction of any guaranteed coverage, path coverage or otherwise. The problem is, if one is after some form of path coverage, in the face of astrononomically many path (or infinitely many), one in practice covers approximately 0% of all paths, even if one invests is a huge amount of test cases. Zero percent sounds worse than it maybe is; after all, it's not coverage one is after, it's about getting the software right, resp. spotting errors or faults (and then repairing them). A particular defect or its symptom may well not be reached by exactly one path, which one hits or misses, but by very many. Besides, there are heuristics one could use, one could check standard corner cases inherent in the input data or, if one has that, corner cases in the *specification* of the unit one tests. That then goes in the direction of *white box* testing, since the test selection is done on the input-output data, and that can be done without having access to the internals of the code or the CFG.

There is another, standard thing one can do, namely working with abstractions.

Abstraction and "static analysis" Abstracting away from details (in a systematic way) allows to cover *all* possible behaviors. The price of that is that one looses precision.

The presentation here presented SE as a way to systematically represent possible paths via path conditions. The representation of the paths is assumed *precise* but collecting exactly the boolean conditions along the way. It's only we run into trouble when solving them. Static analysis characteristically works with techniques like data flow analysis (or more generally abstract interpretation or type system in way that systematicall *approximates* the concrete behavior. One (typically) does not attempt to capture *precisely* which choices of values lead to which paths. Instead, one works with approximations (of the values) but does not attempt to tailor-make the abstractions such that they fit exactly the paths.

In a way, the treatment in symbolic execution works on abstractions, as well. The values of the input space are carved up. As far as the values for y are concerned, they are grouped into two classes: all the values where $y = 10$ and all the values $y \neq 10$. One can see that as having two abstract values for y, one consisting of $\{10\}$ and one of the set $\mathbb{N} \setminus \{10\}$. That they are represented "symbolically" with "formulas" or constraints is more a matter of perspective. But SE is based on the idea that the abstraction is sculpted by the need to "steer" the abstract execution along all possible paths (at least those which are realizable), and that works fine as long as there are only finitely many such path.

What an approximating analysis on the other hand does is to assume that it can go *either way*, but without remembering which way it goes, just running the analysis approximately (the technical terms is that the analysis is "path insensitive"). There is more that distinguishes data flow analysis from SE. One is that often the purpose is different. In data flow analysis, the purpose is often not to split up the input of a procedure to get good coverage for testing (though it's a legitimite goal as well). Instead, one analyses (often in the context of a compiler) other aspects of the code. Therefore, even if one is as radical as representing variables like x and y just by the knowledge that they are integers, one typically adds additional information related to what one is interested in (for live variable analysis, some information about when the variables is assigned to, for analysis of nil-pointer problems, when pointer variables get a proper value etc). And typically that is done also not just for input variables of a procedure, but for all variables or other entities one is interested to analyze. In any case, static analysis like data flow analyses are typically not path-sensitive (as explained). Path sensitivity is not fundamentally forbidden, it's just too expensive to do in many applications. As a consequence, such analyses are less precise, i.e., more approximative. In doing so, problems with undecidablity may disappear thanks to working with abstractions, and loops no longer pose a problem, at least not as serious as for SE.

One way to see analyses like data flow analysis is not to work with abstractions that exactly cover all combinations of "true" and "false" for all encountered conditions. The abstraction is done *independent* from that. In the simplest case (with the most radical abstraction), one could completely ignore the concrete value (perhaps just abstracting it into its type, like `int`). Obviously, when encountering a condition mentioning the comparison $y = 10$, the analyser would not know if the run goes left or right in that case. One might also split into 3 different abstract values, maybe $\{\mathit{negative}, 0, \mathit{positive}\}$, hoping that this is a good choice, but the choice is independent from the conditions in the program.

The *borderline* between SE and static analysis is, however, not clear cut. For instance, one could do the following: one can replace constraints beyond the capabilities of the chosen SMT solver (like the one involving x^3) by a constraints in *linear arithmic*. Sometimes one can approximate non-linear constraints by linear one. That way, one can no longer have the exact correspondance between the paths and solutions of the path constraints, therfore it becomes a but like (other) static analyses.

So, isn't SE not a static analysis, as well? It sure is, in that it analyses statically the code. Why it's presented here as being slightly different is its motivation: it's part of a more advanced *testing* approach, which is not a static analysis. Testing is *run-time* or *dynamic* analasys. But it's fair to see SE in the presentation here as a static analysis technique used to improve the run-time technique of testing.

7.3 Concolic testing

"Concolic" is a portmanteau work, mixing together the words "concrete" and "symbolic". Another name for the technique is also DSE, **dynamic symbolic execution**. The presentation here covers the approach as realized by the Dart-tool, which introduced the idea [2]. Since it's introcution, the tool and technique evolved further, as well as the acronyms of the tool(s).

It rests combination of two techniques: a) Random testing, which works with dynamic executions involving concrete values and b) symbolic execution, which works statically and with symbolic constraints and formulas. The slogan of the approach is:

Execute dynamically & explore symbolically

In the following we show in a series of figures, how Dart combines random testing and symbolic execution into a *concolic execution* framework. In the slide versions, the exploration of the different path is shown stepwise, in overlays, which illustrate the interplay between the dynamic execution and the symbolic exploration of alternatives . The overlays are not reproduced here.

The example is taken from Section 2.5 from Godefroid et al. [2] and shows how to handle the program from before (Listing 7.1), which involves non-linear constraints. The non-linear constraint there is meant as an example of a constraint that can't be handled by the chosen SMT solver. Often those focus on decidable theories (like linear constraints).

Also standard *over-approximation* techniques ("predicate abstraction") may not be able to precisely analyze a program like that. They would be unable to figure out that a fail state is unreachable taking the path "via the right-hand side" from Figure 7.2d), i.e., unable to pinpoint unrealizable path. The best they would do is that it "may be reachable", thus reporting an error that is actually not possible. The overapprixmation thus leads to *false alarms*. False alarms are problematic if the user drowns in them. The "tester" may have no patience to inspect thousands of warnings, most of which are just false alarms. So, the tool may become unhelpful if the approximation is too coarse. Complex programming structures, especially wild pointer manipulations and spaghetti code, but also *dynamic aspects* such as higher-order functions, dynamic or late binding etc. confuses not just the programmer but also lead to wildely approximative (= unusable) results. Things get worse when adding concurrency to the mix ...

For the example. Figure 7.5 shows a possible first run of the Dart tool. It starts like random testing, picking an random input, say

$$(x, y) = (700, 500)\ . \tag{7.4}$$

This input leads to the path marked in red in Figure 7.5. Of course, picking exactly those two numbers is highly improbable, but picking an x larger than 0 and $y \neq 10$ has a probability of almost 50%. Of course since it's random, Dart may alternatively start off by choosing the input that leads to the path to completions on the right-hand side, which has a probability of likewise 50%. Only the third possible path, stumbling directly across the error by picking $x > 0$ and $y = 10$ is highly unlikely. Anyway, we start as shown in Figure 7.5.

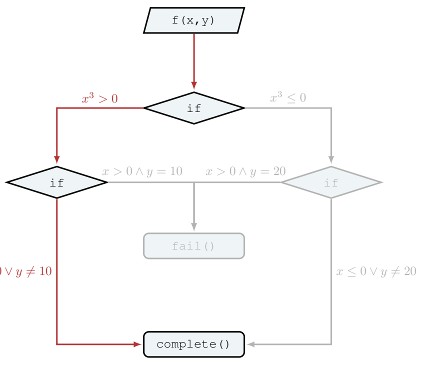


Figure 7.5: Dart (1) (same as Figure 7.2a)

While doing the concrete test run with that input, two boolean conditions have been evaluated to true: $x^3 > 0$ and $y \neq 10$. Those are the path conditions corresponding the path randomly picked. Now, with the goal of path coverage in mind: one should continue with exploring *alternatives*, i.e., explore paths behind the *negation* of those conditions. The negation of the first one is $x^3 \leq 0$. That's a non-linear constraint, i.e., one where a typical SMT solver may chicken out.

So assume Dart does not attempt to do any constraint solving here. Remember the goal: we want to find more or less systematically all paths, but we don't want to overapproximate; we don't want to include unrealizable paths as the might result in false alarms. As we cannot find the *alternative* route at this point in the chosen path by *solving* $x^3 \leq 0$. the only thing we can do at this point is to *use* the path we know that exists as fall-back. That's the path we are currently pursuing, which "solves" the constraint $x^3 > 0$ in having the concrete 700 as one solution.

So we use the *concrete* execution as witness to find one witness solving a constraint we cannot otherwise solve via SMT (more precisely, when we cannot solve its negation, but that amounts generally to the same). In that particular example, we add $x_1 = 700$ as constraint (let's write x_1 when referring to the x in the first run). Now we continue the run with the next conditional. With y picked as 50, the condition $y \neg = 10$ is true. In this

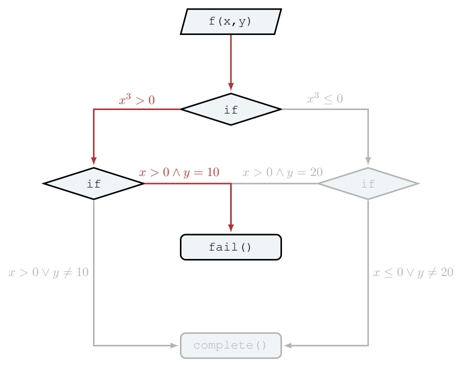


Figure 7.6: Dart (2) (same as Figure 7.2b)

case, the the negation is $y = 10$ which is perfectly solvable (actually: a constraint of that form, equating a variable with a concrete, constant value is a constraint *in solved form*). That's good, so constraint "solving" gave us that $y = 10$ would lead to a *different* path.

So sum up the first run: the randomly generated input from equation (7.4) led to the *concrete execution* from Figure 7.5, and a constraint system of the form

$$(x_1, y_1) = (700, 10)$$

The x_1 is the concrete value in this run, the constraint for y_1 comes from symbolically representing the corresponding alternative in that run (it so happens in the example that the constraint is already in a form ($y_1 = 10$) that has only one solution.

This is the starting point for the *second* run of the method, which is shown in Figure 7.6.

Applying the same method as in the first run, x has the same problem as before, which means we need to use the concrete value 700 as fall-back. That leads to the constraint

$$(x_1 = 700) \wedge (y_2 \neq 10) \ .$$

However, that corresponds to a path already explored. Consequently, after the second run (in this example), *no new inputs are generated.*

If we don't have clear direction (in the form of constraints) what input to take next, we can of course generate a new one randomly. That obviously may result in path already explored. However, in the example, the portion of the graph not yet explored so far is the right-hand side. Sooner or later, the random input generation will pick an input with $x \leq 0$, which explores that part. And actually, it will happen rather sooner than later, let's assume, at iteration n. For concreteness, let's assume the concrete input is

$$(x, y) = (-700, 500) \ .$$

That leads to an execution covering the path from Figure 7.7. The symbolic part chickens out on the first constraint which involves x^3 (besides that the left-hand alternative $x_n^3 > 0$ is already explored), so we have the concrete value $x_n = -700$.

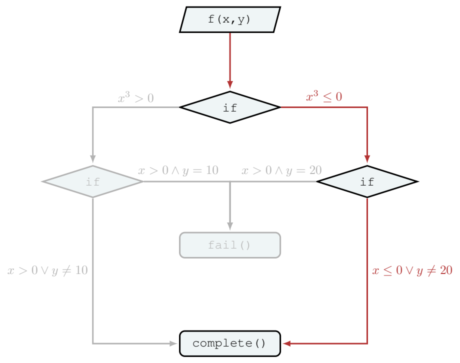


Figure 7.7: Dart (n) (same as Figure 7.2c)

The conditional leads to the additional constraint $x_n > 0 \wedge y = 20$, but that means we have

$$x_n = -700 \quad \wedge \quad x_n > 0 \quad \wedge \quad y = 20 \tag{7.5}$$

which is unsatisfiable. By general reasoning involving the non-linear term x^3, we were aware that this path is unrealizable for any choice of x. The SMT solver may be too weak to draw that conclusion, but at least it will never explore that path, since when the symbolic execution does not work, it relies on *concrete* executions, and those never take that path. So: *no false alarms!*

At that point, we cannot generate new paths any more, all 3 possible paths are covered and the one unrealizable was "covered" insofar that it has been half-symbolically and half-concretely evaluated (see equation (7.5)). So, when figuring out that, the method *stops* generating new tests, having achieved (in this example) the best possible path coverage without generating false alarms.

One can convince oneself, that even with alternative random picks, for instance starting to explore the right-hand side instead of the left hand side as in this illustation, the result would be the same. So with very high probablity (and in short time), the method will achieve that coverage.

Side remark 7.3.1. The example, taken from [2], serves to illustrate in which way the combination of symbolic and concrete execution improved on both plain random testing, symbolic execution, and on approximative methods: it is highly improbably that random testing find the bug, symbolic execution cannot handle the example, and overapproximation give false alarms. Hurrah for concolic execution!

But, on second thought, the example is hand-crafted with the intention to "prove" the superiority of that methods over some competitors. But is it wholly convincing? Well, it worked convincingly enough in the example, in particular stressing the high probablity of covering all realizable paths in a short amount of time.

But that may depend on the (perhaps too cleverly) constructed example. There are two integer input domains: the one for x and the one for y. The one for x is divided 50-50, namely for $x \leq 0$ and $x > 0$. The other domain is *split in an extremely uneven way*: $y = 10$

vs. $y \neq 10$. In both cases the split of the domains correspond to different paths that need to be covered. The SMT solver cannot tackle the *even* split domain for x, as it is written in the form $x^3 \leq 0$ and $x^3 > 0$. The *uneven* split for y, luckily, can be represented by linear constraint and the symbolic treatment can therefore cover the two choices very fast. The even coverage can, with high probability, be covered quite fast by random generation.

If we would have written $y^2 \neq 100 \wedge x > 0$ instead of $y = 10$, the DART method would struggle as well.

So, the example should be read as illustration, in aspects one can hope to improve of the other approaches. Whether it in practice is a step forward can be judged only by applying a corresponding tool to real example programs. Besides that, it also depends on practical issued (which kind of theories should be reasonably covered by the SMT, what data structures does the programming language support, what about external variables and external procedure call etc). The paper [2] reports on experimental evaluation of their approach, providing evidence that the method gives quite added value compared to pure random testing, but they also point out problems of the method in practice

It should also be said, that DART is not the only attempt to improve "stupid random testing" by similar ideas (also before that particular paper). $\square$

Bibliography

[1] Baldoni, R., Coppa, E., D'Ella, D. C., Demetrescu, C., and Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Computing Survey*, 51(3).

[2] Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: Directed automated runtime testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM.

[3] King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

Index