Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2019

Martin Steffen, Volker Stolz

Contents

Chapter 6
Symbolic execution

What is it about?

Learning Targets of this Chapter

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

Contents

6.1 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 clipped in from that presentation. More material may be found in the survey paper [1].

Introduction

- **symbolic** execution: "old" technique [3]
- natural also in the context of **testing**
- **concolic** execution: extension
- used also in compilers
  - code generation
  - optimization

Code example

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

  complete();
}
```

The code does not has any particular purpose, except that it will be used to discuss testing, symbolic execution, and a concept called 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, loops poses challenges for symbolic execution. The problems are similar to the challenges to bounded model checking, which was covered by one of the earlier student presentations. BMC is a technique which shares some commonalities with symbolic execution: both are making use of SAT/SMT solving.

How to analyse a (simple) program like that?

- **testing**
- “verification” (whatever that means)
  - could include code review
- model-checking? Hm?
- symbolic and concolic execution (see later)

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 is considered out-dated by some, but widely used anyway).

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 criterial. 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 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 (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.

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*.

Programs typically contain more control structure than just one single condition. 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 much more to say about symbolic execution as a field, but that's the core idea in a nutshell.

6.1.1 Testing and path coverage

Testing

- maybe **the** most used method for ensuring software (and system) "quality"
- broad field
  - many different testing goals, techniques
  - also used in combination, in different phases of software engineering cycle

- here: focus on

"white-box" testing

- AKA structural testing
- program code available (resp. CFG)

- also focus: *unit* testing

Goals

- detect errors
- check corner cases
- provide high ("code") **coverage**

(Code) coverage

- note: typically a non-concurrent setting (unit testing)
- different coverage criteria
  - nodes
  - edges, conditions
  - combinations thereof
  - path coverage
- defined to answer the question

When have I tested “enough”?

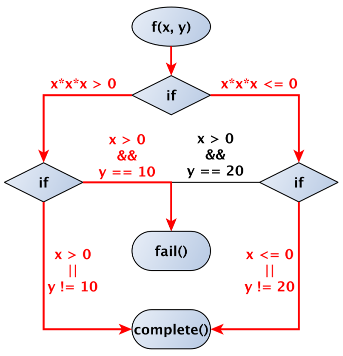
path coverage

- ambitious to impossible (loops)
- note: still not *all reachable states*, i.e., not verified yet

As mentioned earlier, *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.

Path coverage

The picture shows the control-flow graph of the program from the beginning.



In the presentation slides, there is a number of overlays (not reproduced here) that show different paths from the start node till one of the terminal nodes, in connection with the program code. The path are marked red in the picture, and there are 3 path marked that way. There are actually 4 different paths in the CFG, but one of them reaching the failure end state via the route on the right does not correspond to a possible execution. That's easy to see, insofar that would imply a value for x satisfying the constraint

$$(x^3 \leq 0) \wedge (x > 0 \wedge y = 20) \ . \tag{6.1}$$

The corresponding path is thus not colored in red in the picture. The constraint from equation (6.1) is an example of a *path condition* or, synonymously, *path constraint*, namely the one for a path, which happens to be unrealizable as the constraint is unsatisfiable.

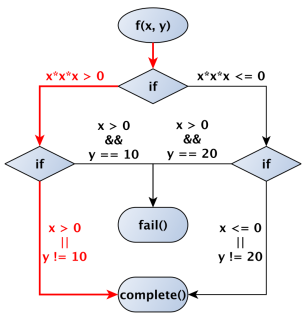
Random testing

- perhaps most naive way of testing
- generating random inputs
- **concrete** input values
- **dynamic** executions of programs
- *observe actual* behavior and
- compare it against *expected behavior*

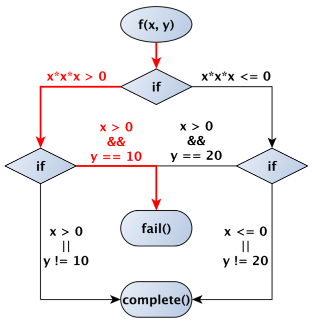
The slide called the approach a "naive" way of testing, but it does mean it has not been used and is being used (and evaluated and compared to other forms, it has been refined etc.) So it has its place.

Random testing

If one applied the randing testing approach to the program from before, one needs to generate random input for the two inputs x and y. Two generated values for that pair could be $(700, 500)$ and $(-700, -500)$, for instance. The red path in the figure shows the path the program takes for the first choice. For the second choice, a diffent path would be chosen, also leading to successful completion.



One path so far missed



As mentioned, there is an unrealizable 4th path, which we should not count that 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 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).

6.2 Symbolic execution

Symbolic execution

- **symbols** instead of concrete values
- use of **path conditions**, aka **path constraints**
- cf. connection to SAT and SMT
- constraint solver computes real values

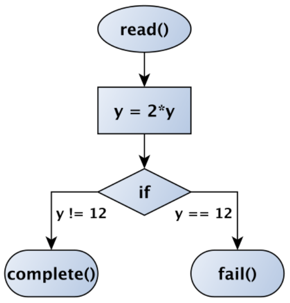
Basically we have introduced the core idea of symbolic execution already earlier. Perhaps it's worth interating that, like in BMC, it's about SMT-solving (not just SAT solving). We are dealing with boolean combinations of constraints over specific domains with specific *theories* (like integers, or arrays, etc.), that corresponds to data types used in the programming language used for the programs we are analyzing. From the presentations about BMC and constraint solving, we also are aware, that theories may easily lead to undecidability of constraint solving. Integers with only addition have a decidable theory (known as Presburger arithmetic). 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" integer, also "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 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. First we finish up with symbolic execution by looking at a super-simple example (but without adding new technical content to the material, it's more like rubbing it in a bit more).

Simple example

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



- in the code: *assignments* not equations (`y := read()`)
- introduce variable `s` for `read()`
- assignments
  - `y := read()` $\Rightarrow y = s$
  - `y := 2*y` $\Rightarrow y = 2s$
- branching point in line 4
  - right: $2s = 12$
  - left: $2s \neq 12$

The code is even simpler than the previous one. The code uses C-like notation where assignment are written using the =-symbol. To differentiate imperative assignment in the programming language from declarative equations used in constraints, the slide writes `:=` for the former. The difference should be clear by 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.

Which input leads to the error?

Constraint solver

$$\text{Solve the path constraint } 2s = 12$$

- child's play: the solution is $s = 6$
- but: requires solver that can do "arithmetic", including multiplication

The point about "multiplication" has been mentioned 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 $2x+17y < z$ 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 inequations (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.*

In summary

Symbolic execution for dummies

- take the code (resp. the CFG of the code)
- collect all paths into **path conditions**
  - big conjunctions of all conditions along each the path
  - each condition b will have
    - one positive mention b in one continuation of the path
    - one negated mention $\neg b$ in the other continuation
- solve the constraints for paths leading to errors with an approriate SMT solver

- works best for loop-free program
- cf. also SSA
- but there is another problem as well (see next)

The remark about SSA may be ignored. SSA stands for static single assignment, a widely used intermediate representation in compilers. It's not the same as path conditions, but shares some commonalities in the treatment of variables. Variables are in most languages not single-assignment, they may be overwritten). However, the SSA format (among other things) introduces "versions" of the source level variables which makes them single-assignment (actually, not really dynamically single assignment, but statically single-assignment). The differentiation betwenn static SA and "real" SA is relevant only when deadling without loops. In loop-free programs, the SSA format transforms the code into some version which is really single assignment. In that way variables become declarative, like variables in a constraint system and the representation of variables for instance in BMC. This has different advantages when it comes to optimization and analysis of the code, which explains the wide usage of that concept. It's outside the scope of this lecture, though.

Complex condition x^3

- non-linear constraint
- in general **undecidable**
- most constraint solvers throw the towel
- for instance: execution stops, no path covered

Coming back to the code example from the beginning of the chapter, we see that this time, the numerical constraints involved are not linear anymore. So, we are definitely leaving the safe ground of decidable theories.

What can one do?

What can one do (beyond throwing the towel and accept that SE won't cover all paths)?

- "static analysis": abstracting
  - cover both path approximately
- theorem proving? one cannot sell that to testers

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 may run into trouble when solving them. By "static analysis" I mean techniques like data flow analysis (or more generally abstract interpretation). Characteristic is there, that one *approximates*. 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 the analysis then 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), though it's not fundamentally forbidden, it's just too expensive to do in many application. As a consequence, they are less precise, i.e., more approximative. Though 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 $\{negative, 0, positive\}$, 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) but 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* analsys. 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.

Concolic testing Concrete & Symbolic = "concolic"

6.3 Concolic testing

Concolic testing

- here following *DART*
- combination of two techniques

Random testing

- concrete values
- dynamic execution

Symbolic execution

- symbols, variables
- static analysis

- other name: **Dynamic symbolic execution** (DSE)

The slogan is

> Execute dynamically & explore symbolically

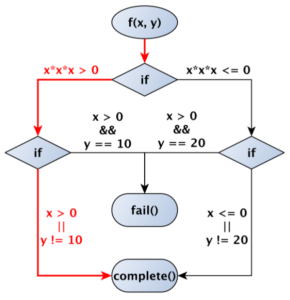


Figure 6.1: Dart (1)

Dart overview

The following slides show how DART combines random testing and symbolic execution into a *concolic execution* framework. In the slides, different runs are shown in a series of overlays. The script version does not show all the overlays step by step. It just shows, for each iteration, one complete path only. The example is taken from Section 2.5 from Godefroid et al. [2]. It shows how DART could handle the program from before, which involves non-linear constraints. Because of that, standard SMT solvers may not not be able to tackle it, since often one restricts to decidable theories (like linear constraints). Also standard overapproximation techniques ("predicate abstraction") may not be able to precisely analyze a program like that. They would be unable to figure out that fail state is unreachable taking the path "via the right-hand side". 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" will 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 imprecise. Complex programming structures, especially wild pointer manipulations and spaghetti code, by also *dynamic aspects* such as higher-order functions, dynamic or late binding etc. confuses not just the programmer but also lead to radically approximative (= unusable) results. Things get worse when adding concurrency to the mix ...

For the example. Figure 6.1 shows a possible first run of the DART tool. It starts like random testing, picking an random input, say

$$(x, y) = (700, 500) \ . \tag{6.2}$$

This input leads to the path marked in red in Figure 6.1. 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 pickig $x > 0$ and $y = 10$ is highly unlikely in the first run. Anyway, we start as shown in Figure 6.1.

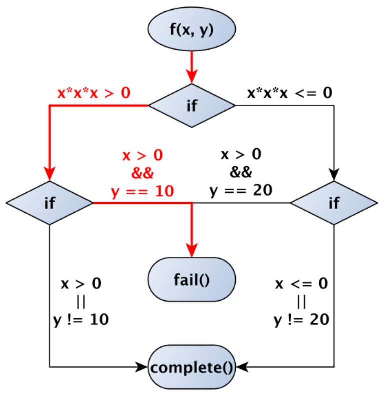


Figure 6.2: Dart (2)

While doing the concrete execution with that input, 2 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 that are 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 chickens 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 $x^3 > 0$. 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 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 (6.2) led to the *concrete execution* from Figure 6.1, 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 6.2.

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

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

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 6.3. 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 $x_n = -700$.

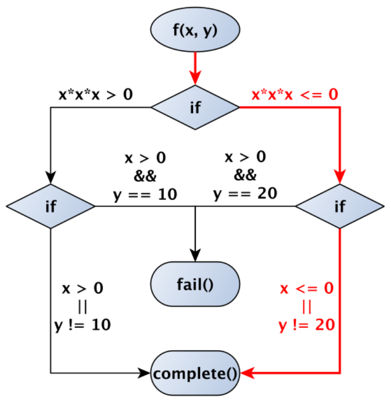


Figure 6.3: Dart (n)

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

$$x_n = -700 \land x_n > 0 \land y = 20 \tag{6.3}$$

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 is 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, the DART method cannot generate new paths any more, it has covered all 3 possible path and the one unrealizable was "covered" insofar that it has been half-symbolically and half-concretely evaluated (see equation (6.3)). 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 achive that coverage.

Afterthoughts to the example 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).

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