INTRODUCTION TO COMPUTER SCIENCE · WEEK 12 · STATION IV

Formal Methods

Machine code in. Formulae out. Solver next. The syntax-to-semantics bridge, made mechanical — within a bound.

WHERE WE ARE

No tool decides what a program does. Every useful tool is an approximation with a stated bound.

TESTING · WEEK 2

Run it on some inputs. Shows presence, not absence. One point per run in a space of $2^{34,359,738,368}$.

THE COMPILER · WEEK 8

Decides everything about the text, nothing about behaviour. A warning is a question about the text in disguise.

TODAY

For *every* input, for *every* path, up to k steps: can the program reach a bad state? A yes comes with the input. A no is a theorem — up to k.

Rice said: choose what to give up. Today we give up everything beyond step k, and get, in exchange, absence.

COOK AND LEVIN, AS ENGINEERING

A machine's computation can be written down as a formula.

Week 3: addition is a circuit of gates. A gate is a formula over bits: *and*, *or*, *not*. So a 64-bit word is 64 variables, and every instruction of week 6 — add, compare, load, branch — is a formula over the bits of its registers.

One step of the machine is one big formula: the new state in terms of the old, depending on which instruction the pc points at. That is what Cook's proof did to prove SAT hard. Here it is a tool.

SMT

Solvers that speak words and memory, not just bits: the theory of bitvectors, the theory of arrays. Underneath, they *bit-blast* to the SAT of last week and run the same clever search. Z3 and Bitwuzla are two of them.

IN THE TERMINAL · ROTOR

A program that divides by zero — for exactly one input.

```
//
ui
ui
ui
x =
*x
re
*x
a =
if
a =
if
re
el
re
}
```

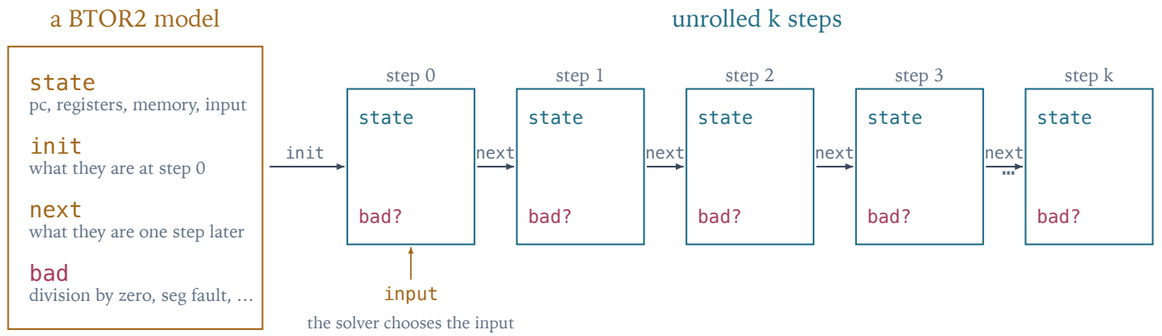
Reads one byte. Subtracts 48, the code of the digit 0. Divides by the result. Type `0` and it divides by zero; type `1` and it exits with 1; type `2` and it divides by zero on the other line; type anything else and it exits with 0.

```
$ make rotor
$ ./rotor -c examples/symbolic/division-by-zero-3-35.c - 0
./rotor: 15161 lines of model formulae generated
./rotor: 237537 characters of model formulae written into examples/symbolic/division-by-zero-3-35-rotorized.btor2
```

Rotor compiles the program with selfie's compiler and writes the model of the RISC-V machine running that binary: every instruction, bit-precisely, as a formula. Linear in the size of the code — it does the same to all of selfie.

THE MODEL

State, init, next, bad.



satisfiable ⇔ some input reaches a bad state within k steps ⇒ the satisfying assignment is the failing input

unsatisfiable ⇔ no input reaches a bad state within k steps ⇒ a theorem, up to the bound k, and nothing beyond it

size of the unrolled formula: O(k × size of the model) · size of the model: linear in the size of the binary

a BTOR2 model, unrolled k steps

```
191 state 5 core-0-pc ; program counter
150 state 71 input-buffer ; uninitialized inp
20070 next 5 191 20064 ; program counter
36027 bad 36026 core-0-division-by-zero
40095 bad 40094 core-0-load-seg-fault
40903 bad 40902 core-0-bad-exit-code ; exit(0
```

States: the pc, the registers, the memory segments, and an input buffer left uninitialised so the solver may choose it. Init: what they are at step 0. Next: what they are one step later — the RISC-V semantics. Bad: twenty-four things the machine must not do.

IN THE TERMINAL · BITME

The solver finds the failing input. Nobody told it to try '0'.

```
$ tools/bitme.py -kmax 120 --use-bitwuzla examples/symbolic/division-by-zero-3-35-rotorized.btor2
bitme bounded model checking: -kmin 0 -kmax 120
0: initializing · 1: transitioning · … · 75: transitioning
76: core-0-division-by-zero
76: vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
76: 150 state 71 input-buffer ; uninitialized input buffer
76: state150-76 = (store … #b0 #b00110000) ← byte 48, the character '0'
89: core-0-division-by-zero … #b00110010 ← byte 50, '2': the second division
106: core-0-bad-exit-code … #b00001100 ← byte 12: exit(0), the flagged exit
```

Bitme walks the steps one at a time and asks about every bad state at each. For the first few dozen the answer is no: the program is setting up its stack, calling `malloc`, calling `read`. Then the division instruction is fetched, and the answer for division by zero is yes — with the input that does it, derived from the formula.

Among all 256 possible input bytes, the solver found the one that reaches the bad state, the way last week's solver found that x1 must be false: not by trying them, by reasoning. And it did it for a compiled RISC-V binary, not for the C.

WHAT A BOUND BUYS

A no up to k is a theorem. It says nothing about k + 1.

TESTING

Visits one state per run. Shows presence. Says nothing about the inputs you did not try.

BOUNDED MODEL CHECKING

Every input, every path, up to k steps. Shows *absence*, within the bound. Says nothing beyond it.

Rice said no tool decides whether a program can divide by zero, and this one does not. It decides whether a program can divide by zero *within k steps*, which is a question about a finite object and therefore decidable — and it pays for the answer in solver time that grows with k. Choosing k is choosing what to give up.

That is formal methods in one sentence, and the workshop around selfie is a set of such choices, each stated: rotor, bitme, babysat, a fuzzer, a second garbage collector, a binary translator — each applied to selfie, and to itself.

STATION IV

Syntax became semantics, and then met the wall.

Take a program, which is syntax, and its execution, which is semantics, and produce a formula whose satisfiability answers a semantic question exactly — within a bound. The formula goes to a solver, and satisfiability is the canonical hard problem.

So a laptop, a small C program and twenty minutes get you to the frontier of complexity theory. That is not a metaphor. It is what just happened on this screen.

NEXT WEEK

A machine that produces programs, fluently. The one thing it cannot supply is the check you just watched. Bring the gate.

BEFORE NEXT WEEK

Recommended exercises.

1. Read the Cost chapter from *Models of Machines* to the end.
2. Generate the rotor model of `examples/symbolic/simple-if-else-1-35.c` and count its state, init, next and bad lines.
3. Run bitme on it with a bound of 100 and report which bad states are reachable and with what input.
4. Change the bound to 20 and explain what a report of *no bad state* now means and does not mean.
5. Write a C* program of your own with a bug that only one input triggers, and let the solver find the input.

NEXT WEEK

Machines: what a large language model mechanically is, why a hallucination is a proof-shaped object that is not true, and the gate that the machine cannot supply.