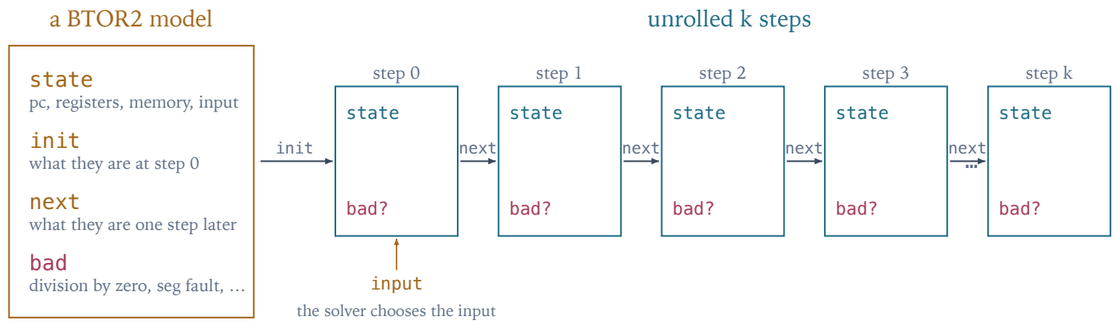
SYSTEMS ENGINEERING · WEEK 10 · STATION IV

Verifying Systems Code

Rotor turns a program into a formula. Bitme asks a solver whether any input reaches a bad state within a bound.

RECAP · ONE SLIDE

A machine state is a formula. A step is a formula. Reaching a bad state in *k* steps is a SAT question.



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: state, init, next, bad, unrolled k steps

Registers and memory are bit-vectors. Each of the fourteen instructions is a function from state to state, expressible in the theory of bit-vectors and arrays. A program's semantics, from the Meaning chapter, is exactly the transition relation of a BTOR2 model: *init*, *next*, and a list of *bad* states.

Unroll *next* *k* times, conjoin one bad state, hand the formula to an SMT solver. Satisfiable means an input exists that reaches the bad state at step *k*, and the satisfying assignment is that input. Unsatisfiable for all *k* up to a bound means no input reaches it within the bound: a theorem.

ROTOR

Rotor generates the model of a RISC-U program with its system calls.

```
$ ./rotor -c examples/symbolic/division-by-zero-3-35.c - 0
… writing division-by-zero-3-35-rotorized.btor2
bad states, among others:
core-0-division-by-zero
core-0-fetch-seg-fault core-0-fetch-invalid-address
core-0-load-seg-fault core-0-store-seg-fault
core-0-brk-seg-fault core-0-bad-exit-code
```

Rotor is selfie's model generator. It compiles the program, then emits the machine as a formula: the registers, the memory segments, the fourteen instructions, and the system calls `read`, `write`, `brk`, `exit` modelled as what the kernel does to the state. Input bytes are the free variables.

The bad states are systems properties: an address outside the mapped segments, a fetch from data, a store into code, a program break that moves backwards, an exit code that is not zero. Memory safety, stated as formulas over the same state the kernel manages. Flags `-Pnosegfaults` and `-Pnoinvalidaddresses` switch families off; the default checks them all.

IN THE TERMINAL

Bitme: unroll, ask, and read the input off the satisfying assignment.

```
$ 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
```

Step by step, every bad state is asked. Seventy-five steps of no while the program sets up its stack and calls read. Then, at step 76, where the division is fetched, yes, and the solver hands back the byte that gets there: the character zero. A failing input, derived, not guessed. Step numbers and format move with the tools; the shape does not. Check before class.

The same run on a segmentation fault looks identical, with a different bad state and an input that computes a bad address. That is the point for a systems engineer: the exceptions your kernel handles are the properties the model checker searches for, and the search is exhaustive up to the bound.

WHAT A BOUND BUYS

Absence, up to k. Nothing about $k+1$. And nothing about threads, yet.

VERSUS TESTING

A test visits one state in a space of 2^{35} bits per machine. A bounded run covers every input and every path to depth k. Presence versus absence; observation versus theorem.

VERSUS RICE

No tool decides whether a program segfaults. This one decides whether it does within k steps, which is a different, finite question. The bound is the approximation, chosen and named, like the timer and the TCB.

VERSUS CONCURRENCY

Rotor models one machine context. Interleavings, week 8's state space, are not in the formula. Modelling them is a research problem with real tools, and not this class's. The formal-methods assignment stays sequential, and says so.

ASSIGNMENTS

rotor-bounds and treiber-stack.

1. **rotor-bounds.** Take a small sequential routine from your systems code: your assembler's parser on a short input, or your `malloc` driven by a program that allocates until the break moves. Write a C* driver that reads its input with `read`.
2. State the property in rotor's terms: which bad state, on which run. Generate the model; run bitme with a bound you choose and justify.
3. Submit the driver, the property, the bound, and either the input the solver found or the bound up to which none exists. One page of notes on what the verdict does and does not establish.
4. **treiber-stack.** `init_stack`, `push`, `pop` on a singly linked list with one shared `top`, using `lr` and `sc` rather than compare-and-swap, so the ABA problem cannot arise. Allocate nodes with your thread-safe `malloc`. Graded by `./grader/self.py treiber-stack`.

GRADING ROTOR-BOUNDS

The grader runs rotor and bitme on your driver with your flags and compares verdict and bound; the exact invocation is in the assignment file. Two weeks for both assignments. The stack is the harder code; the model is the harder thinking.