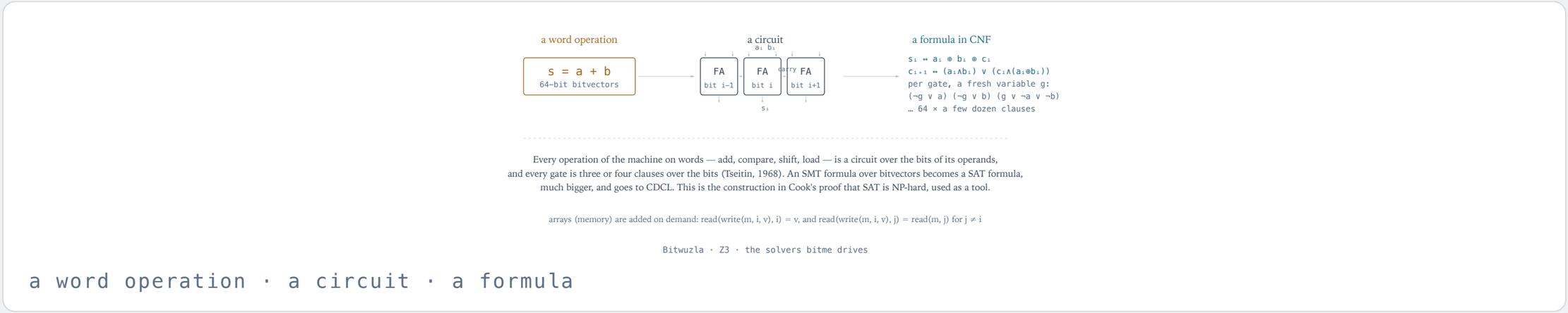
COMPILER CONSTRUCTION · WEEK 12 · STATION IV

Bounded Model Checking

SMT: bitvectors and arrays by bit-blasting. Bitme: the model of week 10 unrolled and handed to the solver of week 11 — and your own extension checked.

THEORIES

A SAT solver speaks bits. A program speaks words and memory. Bit-blasting connects them.



a word operation · a circuit · a formula

Satisfiability modulo theories. A propositional skeleton whose atoms are statements in a theory: bitvectors, with the machine's operations on fixed-width words; arrays, with read and write on an indexed store. SMT-LIB is the notation; Z3 and Bitwuzla read it, and BTOR2 too.

Bit-blasting. A 64-bit word is sixty-four variables; an addition is the adder circuit; every gate is three or four clauses with a fresh variable (Tseitin). Arrays get their axioms on demand. The SMT formula becomes a SAT formula, much bigger, and CDCL runs on it. Cook's construction, as engineering.

IN THE TERMINAL

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

```
$ ./rotor -c examples/symbolic/division-by-zero-3-35.c - 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: vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
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 and, at each, asks the solver about every bad state. Seventy-five steps of no while the program sets up its stack and calls malloc and read. Then, at step 76, where the division instruction is fetched, yes — and the assignment of the uninitialised input buffer is the failing input. Nobody told it to try '0'.

The step numbers move with the versions of rotor and bitme; check before class. The shape does not move: a bad state, a step, an input. And the same run on the students' own extensions is the assignment.

WHAT A BOUND BUYS

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

TESTING · AND THE GRADER

One input per run. Shows presence. The grader's few hundred programs are a few hundred points in the space of week 2 of the introduction class.

BOUNDED MODEL CHECKING

Every input, every path, up to k steps. Shows absence, within the bound, as a theorem. Pays in solver time that grows with k, and says nothing beyond it.

Rice forbade deciding whether a program divides by zero. This decides whether it does so within k steps — a question about a finite object. Choosing k is choosing what to give up: the sound-but-incomplete corner of week 9's table, with the incompleteness made explicit as a number.

Everything in the workshop around selfie is such a choice: rotor and bitme within a bound, the type checker for one property, buzzr complete about the crashes it happens to see. Each stated. Each applied to selfie itself.

ASSIGNMENT

rotor-check: verify your own extension.

1. **A test program.** In C*, using your extension — the for loop, arrays, lazy evaluation, structs — with one input on which it misbehaves: an out-of-bounds index, a division by zero, a wrong exit code, reachable only for some input byte.
2. **A property.** State which of rotor's bad states the misbehaviour is, and the bound within which it is reachable.
3. **The run.** Compile the program with *your* selfie, generate its rotor model, run bitme with your bound. Submit the program, the property, the bound, and the input the solver found.
4. **The check.** The grader recompiles your program with your selfie, regenerates the model, runs bitme to your bound, and compares the verdict and the input.

AND STRUCT-EXECUTION

The other half of the capstone: field access `p->f` as a load or store at the pointer plus the field's offset, types from the entry, the assertions on temporaries. Then use a struct in selfie.c — for the symbol table, if you dare — and self-compile.

Rules as always: selfie.c and the grammar; no warnings; self-compile passes. For rotor-check the submission is a directory with the program and a one-page note.