Project: Iterative SAT-solving Rectifier

Byron Stewart and Anthony Lesik, December 2025

Introduction

This project intends to implement a rectifier similar to Fujita (2012). We will take a sample circuit as a `.blif`, and label it as a Specification. We will make one arbitrary change as a bug, take note of the gate, and call it a Bugged model. Then, we will use CEGIS to identify which gates need to be replaced and with what functions. We want to replicate the result that even though the search space is exponentially large, only a small number of test patterns (~a few hundred) are needed to identify the correct fixes.

There aren't many preliminaries to this report beyond understanding satisfiability proofs and rectification.

Literature Review

Fujita (2012) provides a novel way of rectifying/verifying circuits. It does this using topologically constrained synthesis, which means instead of building a brand new circuit and having some algorithm figure out what fix to apply at certain nets, you replace the gates with parameterized blocks with prior knowledge of where the error is. The circuit topology is passed to the CEGIS algorithm which fills in the parameterized blocks with appropriate gates so that the circuit matches the behavior of the specification. Fujita calls these "local circuit transformations."

The problem is set up as an iterative SAT process. You provide SAT with your implementation circuit (with the parameter variables) as well as the specification. SAT returns the test vectors and variable values that cause the two circuits to mismatch. You add a constraint to the SAT clauses based on the counterexample from the last round, run it again, and continue this process until you reach UNSAT. UNSAT means we found the test patterns that provide full coverage. After we have the test patterns, any valid transformation that passes all the tests we gathered is guaranteed to be correct. By constraining the search space, the problem becomes much smaller and easier to deal with.

Fujita explores different approaches for what the parameterized blocks can be. One is just a controllable inverter, so the decision becomes whether to invert the signal or not. The other is all possible gates (so 16 for 2-input functions). The second option makes the search space much larger, but it's also more powerful for rectification. For our rectification goals, we'll likely implement the full function transformations to maximize our ability to repair faulty circuits.

Source Code description

`circuit_types.py` contains the Circuits class, which takes in a truth table and encodes it as a logic gate. To get the truth table, we wrote a parser in `blif_parser.py`. The core of Fujita (2012) relies on CEGIS to do the iterative synthesis, so we needed to encode our circuit as an SMT problem (`encoder.py`). We used Z3 as the solver, which we finally do in `cegis.py`. We implement it essentially the same as the paper does.

Experiment

Except for the unit tests in each file, all of our benchmarks are circuits taken from the class website. We changed one net, `t3`, in `example2.blif`. Here is the result of a sample run:

```
$ py main.py --impl benchmarks/example2_buggy.blif --spec benchmarks/example2_spec.blif --
fix "t3"

Rectifying circuit: example2.blif
  Gates to fix: {'t3'}
  Total parameters: 4

Running CEGIS...
    Primary inputs: {'a': False, 'b': False, 'd': False, 'e': False, 'f': False, 'g': True,
'h': False, 'i': False, 'j': False, 'k': False, 'l': False, 'm': False, 'n': False, 'o':
False, 'p': False, 'q': False, 'r': False, 's': False, 't': False, 'u': False, 'v': False,
'w': False, 'x': False, 'y': False, 'z': True, 'a0': False, 'b0': False, 'c0': False, 'd0':
False, 'e0': True, 'f0': False, 'g0': False, 'h0': True, 'i0': True, 'j0': False, 'k0':
False, 'l0': False, 'm0': False, 'n0': True, 'o0': True, 'p0': False, 'q0': False, 'r0':
True, 's0': False, 't0': False, 'u0': True, 'v0': False, 'w0': False, 'x0': False, 'y0':
False, 'z0': True, 'a1': False, 'b1': False, 'c1': False, 'd1': False, 'e1': True, 'f1':
False, 'g1': False, 'h1': False, 'i1': False, 'j1': False, 'k1': False, 'l1': False, 'm1':
False, 'n1': False, 'o1': False, 'p1': False, 'q1': False, 'r1': False, 's1': False, 't1':
False, 'u1': False, 'v1': False, 'w1': False, 'x1': False, 'y1': True, 'z1': False, 'a2':
True, 'b2': True, 'c2': True, 'd2': False, 'e2': True, 'f2': True, 'g2': False, 'h2': True}
    Gate t3 sees inputs: [True, True]
    Expected output (from spec truth table): True

================================================
RESULTS
================================================
Status: SUCCESS
Iterations (test patterns): 2
Time: 0.039s

Fixes found:
  t3: CONST0 -> AND
```

Running a circuit against itself returns a 'fix' of the BUFFER gate, or to make no changes. This can be seen with the included `MastrovitoF.blif`, which is unedited from the class website.

```
py main.py --impl benchmarks/MastrovitoF.blif --spec benchmarks/MastrovitoF.blif --fix
"y_5_"
Rectifying circuit: MastrovitoF_q16.eqn
  Gates to fix: {'y_5_'}
  Total parameters: 2

Running CEGIS...
```

```
    Primary inputs: {'a_0_': True, 'b_0_': True, 'a_1_': True, 'b_1_': True, 'a_2_': True,
'b_2_': True, 'a_3_': True, 'b_3_': True, 'a_4_': True, 'b_4_': True, 'a_5_': True, 'b_5_':
True, 'a_6_': False, 'b_6_': True, 'a_7_': False, 'b_7_': True, 'a_8_': False, 'b_8_':
False, 'a_9_': False, 'b_9_': True, 'a_10_': True, 'b_10_': True, 'a_11_': False, 'b_11_':
False, 'a_12_': True, 'b_12_': True, 'a_13_': True, 'b_13_': True, 'a_14_': True, 'b_14_':
True, 'a_15_': True, 'b_15_': True}
    Gate y_5_ sees inputs: [True]
    Expected output (from spec truth table): True

================================================
RESULTS
================================================
Status: SUCCESS
Iterations (test patterns): 2
Time: 9.476s

Fixes found:
  y_5_: BUF (unchanged)
```

Experimentation with multi-fix

We made multi-fix rectification work by refactoring the code to handle a list of nets rather than a single gate. To call rectification, pass in `--fix "gate1,gate2"` with no space. An example is below:

```
py main.py --impl benchmarks/example2_buggy_in_two_gates.blif --spec
benchmarks/example2_spec.blif --fix "t3,u3"
Rectifying circuit: example2.blif
  Gates to fix: {'t3', 'u3'}
  Total parameters: 8

Running CEGIS...
    Primary inputs: {'a': False, 'b': False, 'd': False, 'e': True, 'f': False, 'g': True,
'h': False, 'i': False, 'j': False, 'k': False, 'l': False, 'm': False, 'n': False, 'o':
False, 'p': False, 'q': False, 'r': False, 's': False, 't': False, 'u': False, 'v': False,
'w': False, 'x': False, 'y': True, 'z': True, 'a0': True, 'b0': True, 'c0': True, 'd0':
True, 'e0': True, 'f0': True, 'g0': True, 'h0': True, 'i0': False, 'j0': False, 'k0': True,
'l0': True, 'm0': True, 'n0': False, 'o0': True, 'p0': False, 'q0': True, 'r0': True, 's0':
False, 't0': False, 'u0': False, 'v0': False, 'w0': False, 'x0': False, 'y0': False, 'z0':
False, 'a1': True, 'b1': True, 'c1': False, 'd1': False, 'e1': True, 'f1': True, 'g1':
False, 'h1': False, 'i1': False, 'j1': False, 'k1': False, 'l1': False, 'm1': False, 'n1':
False, 'o1': False, 'p1': False, 'q1': True, 'r1': True, 's1': True, 't1': True, 'u1': True,
'v1': True, 'w1': True, 'x1': False, 'y1': True, 'z1': True, 'a2': True, 'b2': True, 'c2':
True, 'd2': True, 'e2': True, 'f2': True, 'g2': False, 'h2': True}
    Gate t3 sees inputs: [True, True]
    Expected output (from spec truth table): True
    Gate u3 sees inputs: [True, True]
    Expected output (from spec truth table): True

================================================
RESULTS
================================================
Status: SUCCESS
```

```
Iterations (test patterns): 2
Time: 0.068s

Fixes found:
  t3: CONST0 -> AND
  u3: CONST0 -> AND
```

What we Learned

This project solidified our understanding of circuit rectification and how to apply SAT solvers to solve problems. We showed how to declare a implementation, a spec, a miter, and how to solve for SAT. We also had to learn what SMT is and how it's distinguished from regular SAT. We've heard a lot about the applications of SMT proofs, and are glad to have learned how to interface with Z3.

Division of Labor

Anthony: `cegis.py`, `circuit_types.py`, the lion's share of the rest of the code, and devised pseudocode for the project
Byron: `blif_encoder.py`, parts of `main.py` and `encoder.py`, most of the report writing, and multi-fix experimentation

Conclusion and Further Work

We can see that the rectifier works as intended in the circumstances we have tested, including multi-fix circumstance the original paper doesn't look at. Further, we have noted that the number of iterations is small compared to the search space.

Right now, we are limited to 2-input logic gates. While n-input logic gates can be modeled with 2-input logic gates, there are some issues that make this less than ideal. Notably, by having internal nets that do not exist on a real n-input logic gate, we may compute useless ATPG tests later in the verification pipeline.

We transformed everything to SMT because this is what Fujita does and it made it possible to use the Z3 Python package. However, we are curious what differences would remain if we used a Boolean SAT solver. Would the different performance come from the different efficiencies of the SAT solvers, or is it able to do some optimization because it's a strictly boolean problem?

Works Cited

M. Fujita, "Toward Unification of Synthesis and Verification in Topologically Constrained Logic Design," in *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2052-2060, Nov. 2015, doi: 10.1109/JPROC.2015.2476472.