Exercise: Proving circuit equivalence using symbolic model checking

2007-05-22 Martin Fränzle, Andreas Eggers

1 Overview

The goal of this assignment is to strengthen your understanding of Tseitin transformation and its application in the context of circuit equivalence verification.

While working on the tasks of this exercise you will therefore write a tool that

- parses the netlists of two digital circuits,
- generates a constraint system that is satisfiable iff these two circuits are different (i.e. there exists an input such that their outputs differ),
- transforms the resulting formula using the Tseitin transformation presented in the lecture,
- writes that CNF to a file in the DIMACS format.

Then you will use the Boolean SAT solver zChaff to check the CNF for satisfiability. Provided that all transformation steps are implemented correctly then the circuits are equivalent iff no such solution can be found.

In order to test this tool, you will build digital circuits using KTechlab and a small converter that generates the netlists from KTechlab's XML format.

2 Netlist format

The circuit shown in figure 1 was drawn with KTechlab. In the figure, you can see logical gates, "external connectors" and the connections between these blocks.

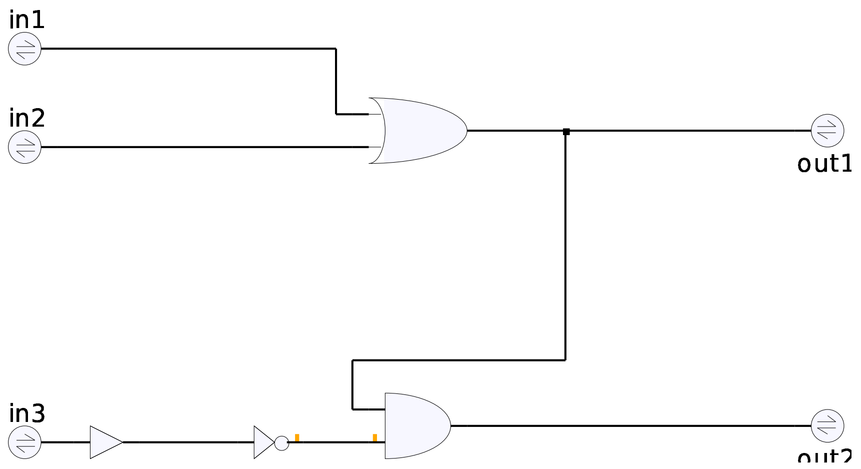


Figure 1: Example circuit

The corresponding XML file was then converted, resulting in the following netlist:

```
# Items
3 = and(2, 1)
5 = 4
CONN: "in1" = 6
CONN: "in2" = 7
```

```
CONN: "out2" = 8
CONN: "in3" = 9
CONN: "out1" = 10
12 = not(11)
15 = or(14, 13)

%

# Connectors
7 = 13
1 = 12
2 = 16
15 = 16
16 = 10
5 = 11
8 = 3
9 = 4
14 = 6

%

# Nodes
# node: 16
```

All lines starting with "#" are comments and can thus be ignored by your parser. The file has three sections: "Items", "Connectors", and "Nodes", which are seperated by a "%". The "Nodes" section contains only comments (stating which of the variables were generated from nodes, i.e. forks in the circuit) and can thus be ignored, too. All logical gates that occur in the circuit are given in the "Items" section. The operators that are supported are: {not, and, or, nor, nand, xnor, xor}. All these operators have their usual semantics. Except "not" all these operators are required to be binary (despite the possibility to create n-ary versions in the GUI). Variables are denoted by strictly positive integers. Let *op* be a binary operator and a, b, c be variables (e.g. 1, 2, 3), then $a = op(b, c)$ denotes that *op* is applied to the arguments b and c and the result given to a.

The external connector elements that are shown in figure 1 are translated to lines starting with "CONN: " followed by the name that was given to the connector in quotation marks. They are then mapped to a variable that follows after the "=".

Buffer elements that have the semantics of the identity function are written as $a = b$ with a, b being variables. The second section consists entirely of such pairs of equalities. These result from the connection lines in the circuit.

Task 1:

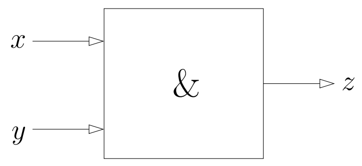
Open KTechlab and create a simple circuit just to become familiar with the tool. Save the circuit to a file and call the converter `circuit2netlist` with the circuit file as the only parameter. The resulting netlist will be written to standard out and can thus be redirected into a file or copy pasted into an editor window.

Task 2:

Write a parser that reads the netlist format as described in this section and generates an internal data structure that is suitable for the following tasks.

3 Circuit Equivalence

In the lecture you have seen examples for propositional formulae that formalize the invariants of logical gates, e.g.



$x \wedge y \Leftrightarrow z$

The conjunction of these local invariants then corresponds to the behaviour of the circuit.

Two circuits are equivalent iff their outputs are equivalent for all possible inputs. You have seen the formalization of this property in the lecture as well.

Task 3:

Write a tool around your parser that allows to read in two netlist files. Assume that external connection blocks with the same name are intended to express the same input / output for both circuits. Add some form of input (e.g. a small file or interaction) such that inputs and outputs can be distinguished. Let your tool then create a formula E that is satisfiable if and only if the circuits are *different*. In order to be able to debug the tool and to trace what happens, a simple output functionality should be implemented as well.

Task 4:

Convert the formula E into CNF by application of the ruleset:

$$\begin{aligned}
(\phi \wedge \psi) \vee \chi &\equiv (\phi \vee \chi) \wedge (\psi \vee \chi) \\
\neg(\phi \wedge \psi) &\equiv (\neg\phi) \vee (\neg\psi) \\
\neg(\phi \vee \psi) &\equiv (\neg\phi) \wedge (\neg\psi) \\
\phi \Rightarrow \psi &\equiv (\neg\phi) \vee \psi \\
\neg\neg\phi &\equiv \phi
\end{aligned}$$

Task 5:

Create Tseitin transformations of the local gate invariants that you used in task 3. Extend your tool in such a way that instead of E now a formula F is generated using these Tseitin transformations of the local invariants that is satisfiable iff the circuits differ. F must also be in CNF.

4 Boolean SAT solving with zChaff

zChaff[1] expects a CNF encoded in the DIMACS format as its input. This format is easy to parse and easy to write:

Assume you want to encode the formula

$$(x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3).$$

The formula consists of two clauses and four variables. You thus write a preamble

```
c My litte CNF example file. This is a comment.
p cnf 4 2
```

The number of variables (4) is given before the number of clauses (2).

Thereafter you write one "0"-terminated line (which may span over more than one "physical" line) per clause of the CNF:

```
1 -2 4 0
1 2 3 0
```

Variables are denoted by integers. The negative literal $\neg x_2$ in the first clause is denoted by -2.

Running zChaff on this file yields:

```
zchaff test2.cnf
Z-Chaff Version: Chaff II
Solving test2.cnf ......
3 vars set during preprocess;

c 2 Clauses are true, Verify Solution successful.
Instance Satisfiable
1 2 3 4
```

and some additional statistics. The solver has found a solution that satisfies the CNF. This solution is given in the same format as the input clauses are: 1 2 3 4, which means that all variables have been set to *true*. In case one of them had been set to *false* it would have been printed with a "-" in front of it.

Task 6:

Write a DIMACS output routine for your tool such that you can give the generated CNFs from tasks 4 and 5 to zChaff.

5 Application to circuits

Task 7:

Generate a circuit in KTechlab that performs the addition of two unsigned 8 bit integers. Regard the 16 bits as inputs and name them "a0" ... "a7" for the first number and "b0" ... "b7" for the second number (starting at 0 with the least significant bit). The output bits shall be "s0" ... "s7" and "c" for the carry (also with 0 the LSB). Agree with another group to use different approaches, e.g. restricting yourself to a certain set of gates. Exchange your circuit files with that other group and run your tool in order to check the equivalence of the circuits. Compare the runtimes for the naive CNF and the Tseitin transformation.

[1] `http://www.princeton.edu/~chaff/zchaff.html`