TECHNISCHE UNIVERSITÄT CHEMNITZ

Technische Universität Chemnitz
Fakultät für Elektrotechnik und Informationstechnik
Professur Schaltkreis- und Systementwurf
Prof. Dr.-Ing. Heinkel

IMPLEMENTING A NETLIST EQUIVALENCE CHECKER USING A SAT APPROACH

Manual

Practical course for the lecture “EDA Tools II”

Contents

1 Introduction

1.1 Aim of the practical course

In the EDA Tools II practice labs, you shall get familiar with how to formally check two given netlists for equivalence using a SAT-based approach. The method shall be implemented in a C++ program that reads in two netlist files in a specified format. If the netlists are functionally equivalent, the program shall display “OK”. If not, it shall print an input assignment which leads to different output values (*counter example*).

1.2 Previous knowledge

It is expected that you are familiar with basic C++ syntax, including the usage of `std::vector` and `std::map` (accessing, adding, deleting and modifying items).

Furthermore, basic knowledge about formal methods is required as taught in the first lecture units in EDA Tools II. Therefore, practice labs usually do not start before mid of May.

1.3 Preparing and using the environment

The following information is valid for PCs in the SSE lab (e. g. hokus.etit.tu-chemnitz.de) that you can also access using remote login (see separated description in OPAL). When logging in from remote, you will basically use a pre-configured Eclipse IDE for C++. Template files and several test netlists are preconfigured as well.

To start working, log in and open a terminal. If you login from remote, make sure that X11 forwarding is enabled (e. g. Xming). Please refer to the document CA_Login_external.pdf for details on how to login from remote.

Type the following command:

```
/home/4alltmp/EDAT2/eclipse_sat
```

Be patient – this might take several minutes! This will create a working copy in ~/EDAT2_SAT. The Eclipse IDE will appear then, as shown in figure 1 (the `sat.cpp` template file has already been opened there).

1.4 Using Eclipse

Figure 1 shows the main Eclipse window after starting. There are several areas (also shown in fig. 1):

- *Text editor:* To edit/extend the provided file `sat.cpp`.

- *File browser:* Displays the files in the project. Most important: `src/sat.cpp` (the to-be-extended C++ program) and several test files in `netlists`.

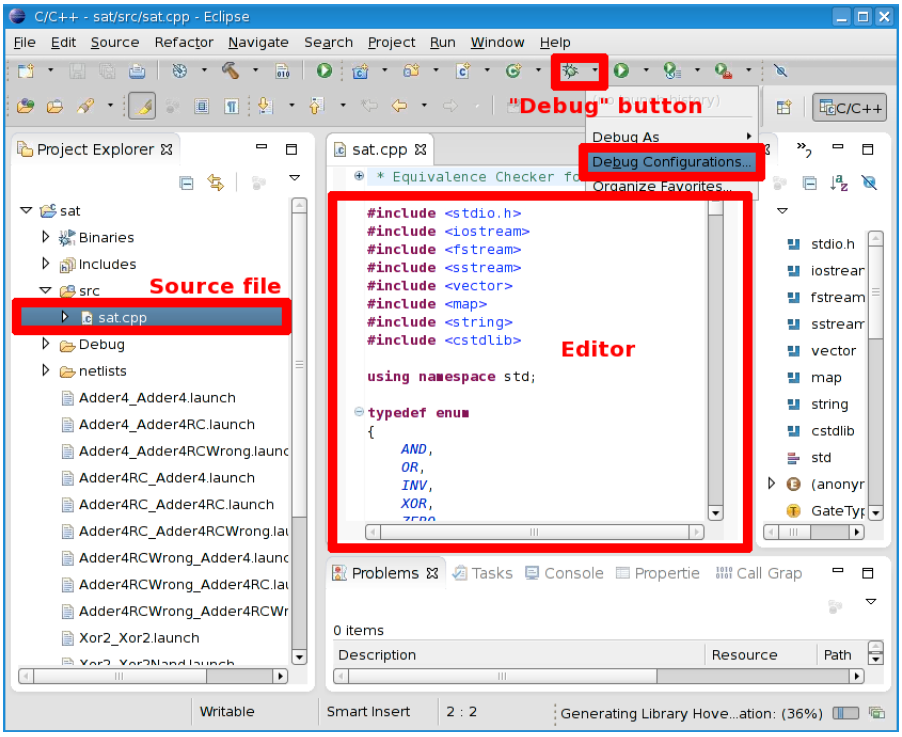


Figure 1: Eclipse IDE.
Use the *Debug* dropdown menu to launch the program / start debugging.

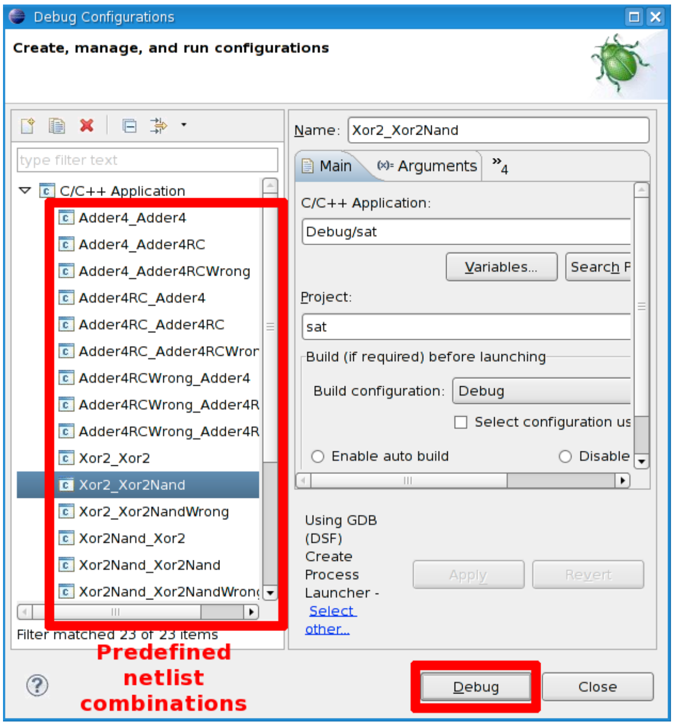


Figure 2: Eclipse Debug Configurations.
Choose the desired configuration and then click *Debug*. E. g. select *Xor2_Xor2Nand* to invoke the program with the arguments `xor2.net` and `xor2_nand.net`.

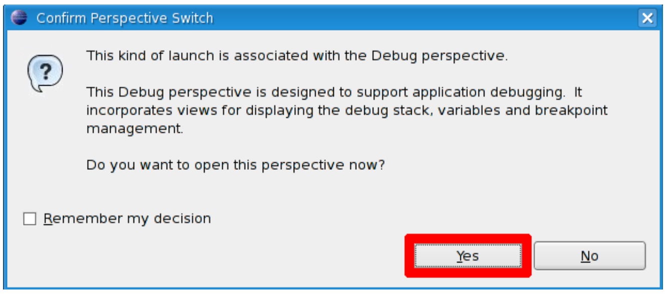


Figure 3: Confirm perspective change. Click *Yes* here.

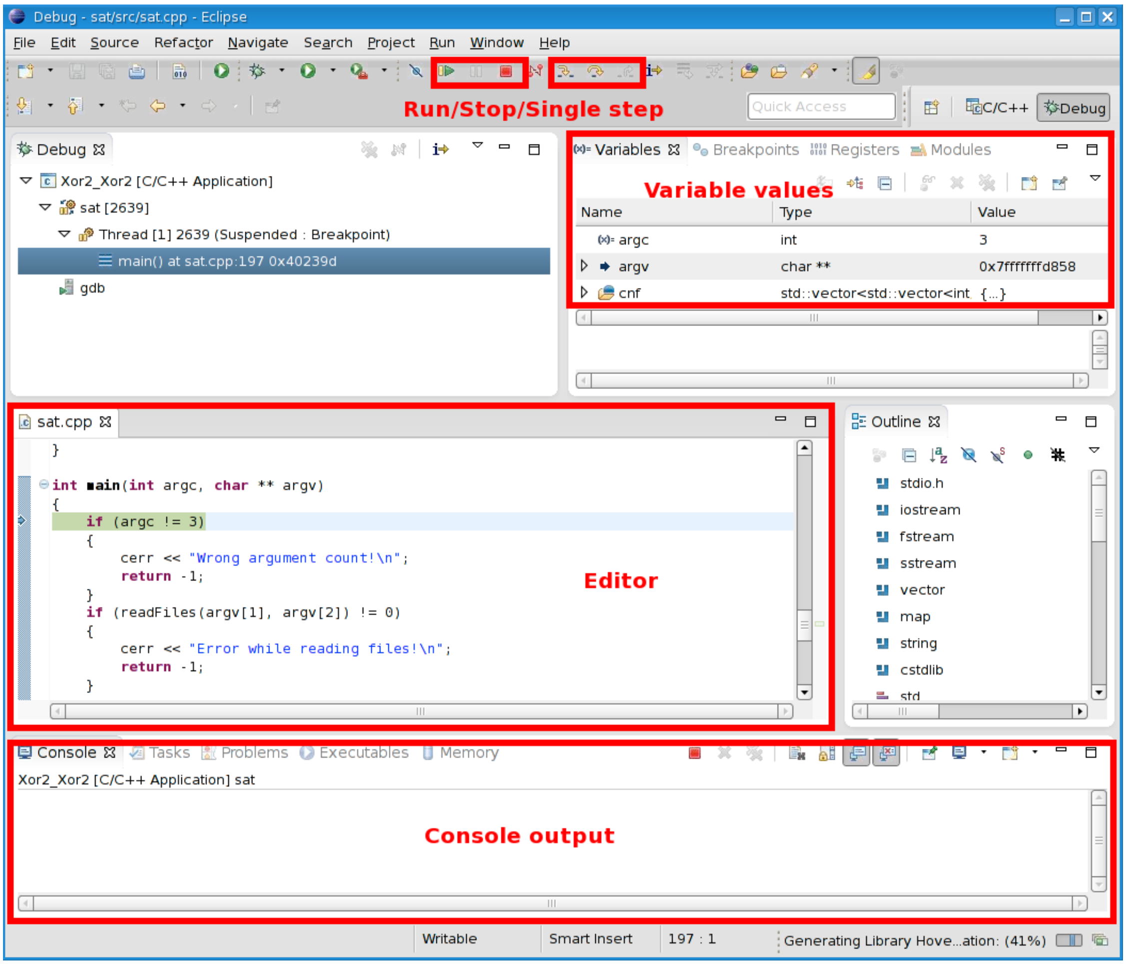


Figure 4: Eclipse Debug Perspective.
In the top bar, there are buttons to control the execution of your program (Run/Stop/Single Step/Step over). You can inspect all variable values in the top-right area. In the editor area, the line that will be executed next is highlighted.

The provided file `sat.cpp` will compile without errors (if not modified). To run it, click the arrow next to the debug button (see fig. 1), select *Debug configurations*, select your desired netlist combination (see fig. 2) and click on *Debug*.

If a window appears whether to change the perspective (see fig. 3), click *Yes*.

Note: Your program should work with any of the predefined netlist combinations (see fig. 2). For simplicity, start with the *Xor2_Xor2Nand* configuration. This will launch your program with the arguments `xor2.net` and `xor2_nand.net`.

1.5 Task

Your task is to write a program that reads in two netlist files (combinational logic) and formally check these for equivalence. If equivalent, print *Equivalent*. If not equivalent, print a counter example.

The theory will be explained in section 2 in this manual, an example is provided in section 3.

Teams of two students are allowed. To reduce workload, a C++ template file is provided (automatically loaded if you use the preconfigured Eclipse IDE). The template file will take care of reading the netlist files and creating data structures. Additionally, several netlist files are provided for test. All provided files are also available in OPAL. Refer to section 4 for details.

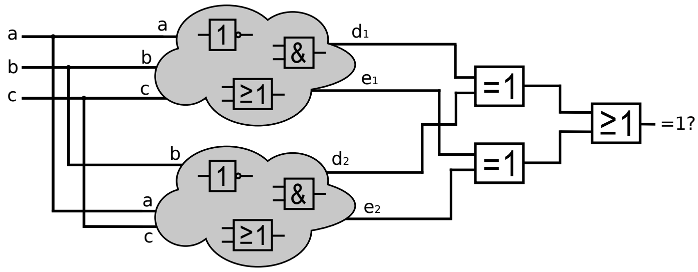


Figure 5: Sample miter circuit.

2 Theory

The idea behind this SAT-based method is to formally check if there is the **possibility** that **at least one output** of the two circuits has **different** values for the **same input assignment**.

In other words: To check for equivalence, you need to **prove** that **all outputs** have the **same value** for **all possible input assignments**. If this cannot be proven, the circuits are not equivalent.

To achieve this, the following steps are necessary:

1. Create an appropriate miter circuit,
2. Convert the miter circuit to a formula in CNF, and
3. Check the CNF for satisfiability.

If the circuits are not equivalent, print an according counter example (i. e. input assignment that leads to different outputs).

2.1 Miter circuit

First, we want to create an "imaginary" circuit that combines both netlists. Such a circuit is called *miter*. It has exactly one output. If it can be proven that this output is zero for all input assignments, the netlists are equivalent. Otherwise, the netlists are not equivalent.

A sample miter circuit is displayed in fig. 5. Let's discuss the details of such a miter circuit.

To ensure that the input assignment is the same for both circuits, we can simply connect inputs that have the same name.

To check if a particular output port has the same value for both netlists, we connect the output ports by using an XOR gate. If the signals differ, the XOR output is 1, otherwise 0. We have to do this for each output.

In case of multiple circuit outputs, we have to find a way to combine all XOR outputs. We do this by using an OR gate. If at least one pair of the circuit outputs differs, the OR output will be 1, otherwise 0.

Finally, assign signal names to all unnamed signals.

Now we have a so-called *miter circuit*. If we can prove that the only output of the miter (the OR output) is 0 for *all* possible input cases, the two circuits are equivalent. If it is possible that the OR output becomes 1, the two circuits are not equivalent.

2.2 Conjunctive normal form

Now, we create a CNF from the circuit. We start with an empty CNF. For each gate, we add its *characteristic function* to the CNF. Additionally, we want to check if the circuits differ, therefore we set the output signal to 1. We do this by adding the according characteristic function of "one" to the CNF. Moreover, we have to connect the inputs of the netlists.

To successfully check netlists for equivalence, the CNF must contain the following:

- Characteristic function for each gate in each netlist,
- Connection of each input pair,
- XOR for each output pair,
- OR to combine all XOR outputs, and
- Set the OR output to "1".

2.2.1 Characteristic functions

A *characteristic function* is a *representation* of a gate as boolean formula. It takes *all* variables (inputs and outputs) as function parameters. The function result shall be one if the parameters represent a valid variable assignment for that gate.

Let's use the AND gate as example. The AND function is defined as follows:

$$c = f_{\text{AND}}(a, b) = a \wedge b$$

Now, we create a Karnaugh map which takes both the inputs *and* the output as parameters. Since the characteristic function shall result in 1 if its parameters represent a valid assignment, we mark the valid assignments with 1, the others with 0. These are the valid assignments:

- $a = 0, b = 0, c = 0$
- $a = 0, b = 1, c = 0$
- $a = 1, b = 0, c = 0$
- $a = 1, b = 1, c = 1$

And these are the invalid assignments:

- $a = 0, b = 0, c = 1$
- $a = 0, b = 1, c = 1$

- $a = 1, b = 0, c = 1$
- $a = 1, b = 1, c = 0$

This is the resulting Karnaugh map:

C \ A B	0 0	0 1	1 1	1 0
0	1	1	0	1
1	0	0	1	0

The characteristic function (in CNF) can be easily derived (as done in the exercises):

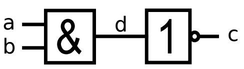
$$fc_{\text{AND}}(a, b, c) = (a \vee \overline{c})(b \vee \overline{c})(\overline{a} \vee \overline{b} \vee c)$$

Preparation task: Create the characteristic functions in CNF for the remaining gate types:

$$\begin{array}{llcr}
b = f_{\text{NOT}}(a) & = \overline{a} & , & fc_{\text{NOT}}(a, b) = \\
b = f_{\text{EQUIV}}(a) & = a & , & fc_{\text{EQUIV}}(a, b) = \\
c = f_{\text{OR}}(a, b) & = a \vee b & , & fc_{\text{OR}}(a, b, c) = \\
c = f_{\text{XOR}}(a, b) & = a \oplus b & , & fc_{\text{XOR}}(a, b, c) = \\
a = f_{\text{ZERO}} & = 0 & , & fc_{\text{ZERO}}(a) = \\
a = f_{\text{ONE}} & = 1 & , & fc_{\text{ONE}}(a) =
\end{array}$$

2.2.2 Deriving a CNF from a netlist

To create a CNF from a netlist, we introduce names for all internal signals and concatenate the CNFs of all individual gates. Assume we have the following netlist:



Here, we introduced *d* as internal signal. The corresponding CNF is:

$$CNF_{netlist} = (a \vee \overline{d})(b \vee \overline{d})(\overline{a} \vee \overline{b} \vee d)(d \vee c)(\overline{d} \vee \overline{c})$$

... which is the result when we concatenate the characteristic functions of the AND and OR gate.

2.3 Solving the CNF

2.3.1 Setting a variable in a CNF

When we apply the Davis Putnam algorithm (next section), we have to be able to set a variable to a specific value and generate the simplified CNF afterwards.

Example task: Set c to 0 in the following CNF:

$$CNF = (a \vee b \vee \overline{c})(\overline{a} \vee \overline{b} \vee \overline{c})(\overline{a} \vee b \vee c)(a \vee \overline{b} \vee c)$$

When setting c to 0, we have to check every clause that contains either c or $\overline{c}$. All clauses that contain $\overline{c}$ can be removed from the CNF. Why? We know that $\overline{c}$ will be 1 when $c = 0$. Therefore, the whole clause will evaluate to 1, e.g., $(a \vee b \vee \overline{c}) = (a \vee b \vee 1) = 1$. Therefore, this clause can be removed.

In clauses containing c, we have to remove the literal c from the clause, since we know that c will evaluate to 0 when $c = 0$. E.g., $(\overline{a} \vee b \vee c) = (\overline{a} \vee b \vee 0) = (\overline{a} \vee b)$.

The resulting CNF for the example task is:

$$CNF_{c=0} = (\overline{a} \vee b)(a \vee \overline{b})$$

2.3.2 The Davis Putnam algorithm

The Davis Putnam algorithm takes a CNF as input and checks for satisfiability. This is the basic Davis Putnam algorithm:

```
def DP(cnf):
    # heuristics
    repeat
        apply pure literal rule to cnf
        apply unit clause rule to cnf
    until heuristics not applicable anymore

    # terminal conditions
    if cnf is empty:
        terminate algorithm, solution found
    else if cnf has empty clause:
        # no solution possible
        return
    else:
        # backtracking
        variable = choose variable from cnf
        cnf0 = set variable to zero in cnf
        DP(cnf0)
        cnf1 = set variable to one in cnf
        DP(cnf1)
```

The algorithm is implemented recursively. In each recursion step, the algorithm tries to simplify the CNF using the *pure literal* and *unit clause* heuristics. After that, terminal conditions are checked: An empty CNF means that a solution has been found, while a CNF with at least one empty clause means that no solution exists. If the terminal conditions do not apply, we have to backtrack (i. e. choose a variable and guess its assignment).

2.3.3 Pure literal rule

The pure literal rule is a heuristic that can speed up algorithm runtime. Consider the following CNF:

$$sampleCNF1 = (a \vee b \vee \overline{c})(\overline{a} \vee \overline{b})(\overline{b} \vee \overline{c})$$

In this CNF, the literal c only exists in its negated form, $\overline{c}$. In this case we know that the only possible value of c is 0. The resulting CNF is:

$$sampleCNF1_{pure} = (a \vee b \vee 1)(\overline{a} \vee \overline{b})(\overline{b} \vee 1) = (\overline{a} \vee \overline{b})$$

2.3.4 Unit clause rule

The unit clause rule is another heuristic. Consider the following CNF:

$$sampleCNF2 = (a \vee b)(a \vee c)(\overline{a} \vee \overline{b})(\overline{a})$$

The last clause in this CNF consists of only one literal. Since we know that all clauses have to be fulfilled (including the last clause), we know that we have to set a to 0, otherwise the last clause would not be fulfilled. The resulting CNF is:

$$sampleCNF2_{unit} = (0 \vee b)(0 \vee c)(1 \vee \overline{b})(1) = (b)(c)$$

2.3.5 Terminal conditions

What does it mean when the algorithm terminates and a solution has been found? It means, there is a valid variable assignment for all variables (signals). Since the main output has been fixed to 1, we know that the variable assignment represents a case in which the circuits under test show different behaviour. Therefore, if a solution could be found, we know that the circuits are not equivalent, and we found a counter example.

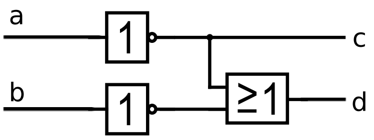
When no solution could be found, we know that there is no assignment that drives the global output to 1. This means that the circuits under test produce the same output for all possible input patterns. Therefore, the circuits are equivalent.

2.3.6 Backtracking

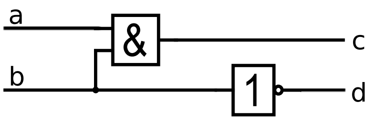
If we cannot apply any heuristic rule anymore, we have to start backtracking. That means that we choose one variable (Hint: for fastest algorithm execution, choose the rightmost variable in the CNF) and try both possible values for that variable – 0 and 1.

3 Example

Consider the following circuits:

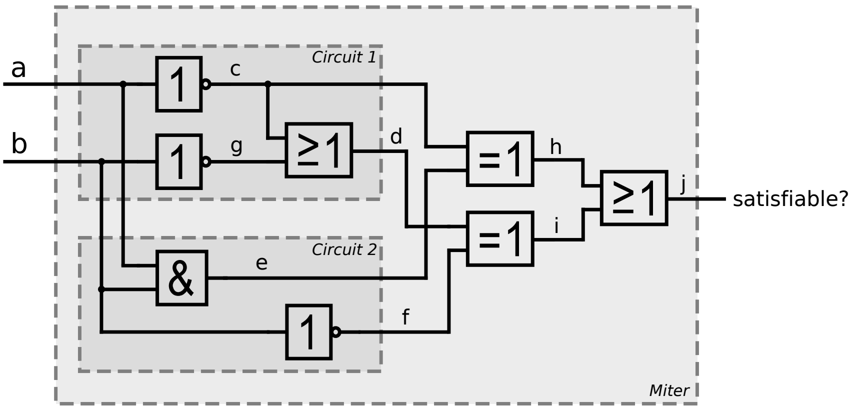


and



3.1 Creating the CNF

We add the miter circuit (connect equally-named inputs, combine outputs by XOR gates, combine XOR outputs by an OR gate, set OR output to “1”) and we assign signal names to all internal signals. Attention: Net numbers in the second netlist have to be modified. The result is one big circuit:



Then, we start with an empty CNF. We append the clauses of the characteristic function for each gate and also assign the static 1 to the output:

$$
\begin{aligned}
CNF_{initial} = (& \\
& \#\ \text{NOT gates} \\
& (a \vee c)(\overline{a} \vee \overline{c}) \\
& (b \vee g)(\overline{b} \vee \overline{g}) \\
& (b \vee f)(\overline{b} \vee \overline{f}) \\
& \\
& \#\ \text{AND gates} \\
& (a \vee \overline{e})(b \vee \overline{e})(\overline{a} \vee \overline{b} \vee e) \\
& \\
& \#\ \text{OR gates} \\
& (\overline{c} \vee d)(\overline{g} \vee d)(c \vee g \vee \overline{d})
\end{aligned}
$$

$$
\begin{aligned}
&\text{\# XOR gates (miter)}\\
&(c \vee e \vee \overline{h})(\overline{c} \vee \overline{e} \vee \overline{h})(\overline{c} \vee e \vee h)(c \vee \overline{e} \vee h)\\
&(d \vee f \vee \overline{i})(\overline{d} \vee \overline{f} \vee \overline{i})(\overline{d} \vee f \vee i)(d \vee \overline{f} \vee i)\\
&\\
&\text{\# OR gate (miter)}\\
&(\overline{h} \vee j)(\overline{i} \vee j)(h \vee i \vee \overline{j})\\
&\\
&\text{\# Final ONE assignment}\\
&(j)\\
)&
\end{aligned}
$$

3.2 Solving the CNF

We call the Davis Putnam algorithm with the initial CNF:

$$DP(CNF_{initial})$$

Step 1

There is no pure literal in $CNF_{initial}$, but because of a unit clause, we can set the variable j to 1. After applying the unit clause rule, we get:

$$
\begin{aligned}
CNF_{j=1} = (&\\
&(a \vee c)(\overline{a} \vee \overline{c})\\
&(b \vee g)(\overline{b} \vee \overline{g})\\
&(b \vee f)(\overline{b} \vee \overline{f})\\
&(a \vee \overline{e})(b \vee \overline{e})(\overline{a} \vee \overline{b} \vee e)\\
&(\overline{c} \vee d)(\overline{g} \vee d)(c \vee g \vee \overline{d})\\
&(c \vee e \vee \overline{h})(\overline{c} \vee \overline{e} \vee \overline{h})(\overline{c} \vee e \vee h)(c \vee \overline{e} \vee h)\\
&(d \vee f \vee \overline{i})(\overline{d} \vee \overline{f} \vee \overline{i})(\overline{d} \vee f \vee i)(d \vee \overline{f} \vee i)\\
&(h \vee i)\\
)&
\end{aligned}
$$

No heuristics are applicable to the resulting CNF anymore and this CNF is not a trivial case, therefore we have to start backtracking. We choose the rightmost variable in the CNF, i. We create two new CNFs, one by setting i to 0, and one by setting i to 1:

$$
\begin{aligned}
CNF_{j=1,i=0} = (&\\
&(a \vee c)(\overline{a} \vee \overline{c})\\
&(b \vee g)(\overline{b} \vee \overline{g})\\
&(b \vee f)(\overline{b} \vee \overline{f})
\end{aligned}
$$

$$
\begin{aligned}
&(a \vee \overline{e})(b \vee \overline{e})(\overline{a} \vee \overline{b} \vee e)\\
&(\overline{c} \vee d)(\overline{g} \vee d)(c \vee g \vee \overline{d})\\
&(c \vee e \vee \overline{h})(\overline{c} \vee \overline{e} \vee \overline{h})(\overline{c} \vee e \vee h)(c \vee \overline{e} \vee h)\\
&(\overline{d} \vee f)(d \vee \overline{f})\\
&(h)\\
&)
\end{aligned}
$$

$$
\begin{aligned}
CNF_{j=1,i=1} = (&\\
&(a \vee c)(\overline{a} \vee \overline{c})\\
&(b \vee g)(\overline{b} \vee \overline{g})\\
&(b \vee f)(\overline{b} \vee \overline{f})\\
&(a \vee \overline{e})(b \vee \overline{e})(\overline{a} \vee \overline{b} \vee e)\\
&(\overline{c} \vee d)(\overline{g} \vee d)(c \vee g \vee \overline{d})\\
&(c \vee e \vee \overline{h})(\overline{c} \vee \overline{e} \vee \overline{h})(\overline{c} \vee e \vee h)(c \vee \overline{e} \vee h)\\
&(d \vee f)(\overline{d} \vee \overline{f})\\
&)
\end{aligned}
$$

We recurse on both CNFs.

Step 2

This is the first recursion step that was invoked in step 1 above. We apply the heuristics to $CNF_{j=1,i=0}$. We find the unit clause (h), therefore, we set h to 1 and get:

$$
\begin{aligned}
CNF_{j=1,i=0,h=1} = (&\\
&(a \vee c)(\overline{a} \vee \overline{c})\\
&(b \vee g)(\overline{b} \vee \overline{g})\\
&(b \vee f)(\overline{b} \vee \overline{f})\\
&(a \vee \overline{e})(b \vee \overline{e})(\overline{a} \vee \overline{b} \vee e)\\
&(\overline{c} \vee d)(\overline{g} \vee d)(c \vee g \vee \overline{d})\\
&(c \vee e)(\overline{c} \vee \overline{e})\\
&(\overline{d} \vee f)(d \vee \overline{f})\\
&)
\end{aligned}
$$

There is no more unit clause or pure literal in $CNF_{j=1,i=0,h=1}$, and this CNF is not a trivial case. Therefore, we take the rightmost variable f and set it once to 0 and once to 1:

$$
\begin{aligned}
CNF_{j=1,i=0,h=1,f=0} = (&\\
&(a \vee c)(\overline{a} \vee \overline{c})
\end{aligned}
$$

$$
\begin{aligned}
&(b \vee g)(\overline{b} \vee \overline{g})\\
&(b)\\
&(a \vee \overline{e})(b \vee \overline{e})(\overline{a} \vee \overline{b} \vee e)\\
&(\overline{c} \vee d)(\overline{g} \vee d)(c \vee g \vee \overline{d})\\
&(c \vee e)(\overline{c} \vee \overline{e})\\
&(\overline{d})\\
&)
\end{aligned}
$$

$$
\begin{aligned}
CNF_{j=1,i=0,h=1,f=1} = (&\\
&(a \vee c)(\overline{a} \vee \overline{c})\\
&(b \vee g)(\overline{b} \vee \overline{g})\\
&(\overline{b})\\
&(a \vee \overline{e})(b \vee \overline{e})(\overline{a} \vee \overline{b} \vee e)\\
&(\overline{c} \vee d)(\overline{g} \vee d)(c \vee g \vee \overline{d})\\
&(c \vee e)(\overline{c} \vee \overline{e})\\
&(d)\\
&)
\end{aligned}
$$

We recurse on both CNFs.

Step 3

This is the first recursion step that was invoked in step 2 above. We apply the heuristics to $CNF_{j=1,i=0,h=1,f=0}$. The unit clause rule can be applied several times:

$$
\begin{aligned}
CNF_{j=1,i=0,h=1,f=0,b=1} = (&\\
&(a \vee c)(\overline{a} \vee \overline{c})\\
&(\overline{g})\\
&(a \vee \overline{e})(\overline{a} \vee e)\\
&(\overline{c} \vee d)(\overline{g} \vee d)(c \vee g \vee \overline{d})\\
&(c \vee e)(\overline{c} \vee \overline{e})\\
&(\overline{d})\\
&)
\end{aligned}
$$

$$
\begin{aligned}
CNF_{j=1,i=0,h=1,f=0,b=1,g=0} = (&\\
&(a \vee c)(\overline{a} \vee \overline{c})\\
&(a \vee \overline{e})(\overline{a} \vee e)\\
&(\overline{c} \vee d)(c \vee \overline{d})\\
&(c \vee e)(\overline{c} \vee \overline{e})
\end{aligned}
$$

$$(\overline{d})$$
$$)$$

$$
\begin{aligned}
CNF_{j=1,i=0,h=1,f=0,b=1,g=0,d=0} = (&\\
&(a \vee c)(\overline{a} \vee \overline{c})\\
&(a \vee \overline{e})(\overline{a} \vee e)\\
&(\overline{c})\\
&(c \vee e)(\overline{c} \vee \overline{e})\\
)&
\end{aligned}
$$

$$
\begin{aligned}
CNF_{j=1,i=0,h=1,f=0,b=1,g=0,d=0,c=0} = (&\\
&(a)\\
&(a \vee \overline{e})(\overline{a} \vee e)\\
&(e)\\
)&
\end{aligned}
$$

$$
\begin{aligned}
CNF_{j=1,i=0,h=1,f=0,b=1,g=0,d=0,c=0,a=1} = (&\\
&(e)\\
&(e)\\
)&
\end{aligned}
$$

$$CNF_{j=1,i=0,h=1,f=0,b=1,g=0,d=0,c=0,a=1,e=1} = ()$$

At this point, no more heuristic rules are applicable. But the current CNF is empty (it has no clauses), and therefore, the algorithm terminates and a solution has been found.

What does that mean? We have an assignment which fulfills the CNF. That means that we have an assignment that will drive the output of the miter circuit to 1. This in turn means that we have found an assignment that shows that both circuits under test are **not equivalent** – the assignment provides a counter example.

The actual counter example can be derived from the assignments that have been made. But only the inputs and outputs are interesting for us. This is the counter example:

Inputs:
```
a:  1
b:  1
```

Outputs of first circuit:
```
c:  0
d:  0
```

Outputs of second circuit:
```
c:  1
d:  0
```

4 Task

Your task is to write a program that implements the flow introduced above. You can use any programming language, however, C++ is preferred, and a C++ template file is provided which contains a function that reads in the netlist files and creates according data structures.

Your program shall implement the creation of the miter and the SAT solver. If the circuits are equivalent, your program shall output **Equivalent**. If the circuits are not equivalent, your program shall print a **counter example** by displaying all input signal names and an according variable assignment.

The output should look similar to these sample (reference) outputs:

```
# ./sat testsuite/xor2.net testsuite/xor2.net
Equivalent!

# ./sat testsuite/xor2.net testsuite/xor2_nand.net
Equivalent!

# ./sat testsuite/xor2.net testsuite/xor2_nand_wrong.net
Not equivalent! Counter example:
Inputs:
a: 1
b: 1

Outputs netlist 1:
f: 0

Outputs netlist 2:
f: 1
```

If you use the preconfigured Eclipse IDE, follow the steps in section 1.4. Otherwise, download the template file and all netlist files from OPAL.

4.1 Input format

Note: If you use the provided C++ template, you can skip this section.

This is a sample input netlist file:

```
10
b a
f
2 b
1 a
10 f

and 1 2 3
inv 3 4
```

```
10 and 1 4 5
11 inv 5 6
12 and 2 4 7
13 inv 7 8
14 and 6 8 9
15 inv 9 10
```

Line 1: Number of nets (here: 10).

Line 2: Names of all input nets.

Line 3: Names of all output nets.

Line 4-6: Mapping from net numbers to port names (for all inputs and outputs).

Line 7: Empty line.

Line 8 to end of file: Gate instantiations in the form: <Gate type> <input nets> <output nets>. Depending on the gate type, there number of parameters may vary. E. g. `and` has three parameters, while `not` has only two parameters.

4.2 Gate types

There are six gate types which are allowed in a netlist:

Gate type	Input ports	Output ports
and	2	1
or	2	1
inv	1	1
xor	2	1
one	0	1
zero	0	1

According constants are predefined in the C++ template.

4.3 C++ template

If you use the provided C++ template, call the compiled program with two netlist files as parameters. The template provides a function to read in the netlist files. Following variables are predefined:

- `int netCount1`, `netCount2`: Number of nets in netlist 1 / 2.
- `std::vector<std::string> inputs1`, `inputs2`: List of input names in netlist 1 / 2.
- `std::vector<std::string> outputs1`, `outputs2`: List of output names in netlist 1 / 2.
- `std::map<std::string, int> map1`, `map2`: Mapping from input/output names to net numbers in netlist 1 / 2.

- `GateList gates1`, `gates2`: List of gates in netlist 1 / 2.
  The type `GateList` is defined as `std::vector<Gate>`, where `Gate` is a struct with members `type` and `nets`. Member `type` will be one of `AND, OR, INV, XOR, ZERO` or `ONE`. `nets` is the list of the actual nets (net numbers) which are connected to this gate.

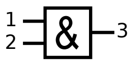
Refer to the comments in the C++ template file for additional details and usage information.

5 Hints

When creating the CNF, use positive net numbers for positive literals and negative net numbers for negative literals.

Since the netlists are described by net numbers (and not letters), you can create a CNF as list of list of numbers, e. g., std::vector< std::vector<int> >. Use positive numbers for positive literals/variables and negative numbers for negative literals/variables.

Consider this netlist:



The corresponding CNF could be represented as:

$$\{\{1,-3\},\{2,-3\},\{-1,-2,3\}\}$$

Do not use net numbers twice.

Example: If you read in two netlists, the first netlist might use nets from 1 to 10, and the second netlist uses nets from 1 to 20. Note that both netlists will be instantiated in parallel in the miter. Therefore, you have to use different net numbers for the second netlist, e. g. by adding a large value to the numbers, e. g. 1000. This would result in nets from 1 to 10 for the first netlist, and nets from 1001 to 1020 for the second netlist.

Do not reuse previously used net numbers for the miter circuit.

When you add the additional miter circuitry, you will need some more nets. Make sure that their net numbers are not already used by the first or the second netlist. E. g., use very large numbers.

Track the current variable assignment in the Davis Putnam algorithm.

In the algorithm depicted on page 10, variable assignments are not tracked. You have to find a way to track variable assignments. Otherwise, you are not able to output a counter example.