VLSI CAD: LOGIC TO LAYOUT: MiniSat Software Package

Rob A. Rutenbar and Chen-Hsuan Lin
University of Illinois at Champaign-Urbana

MiniSat[1] is a SAT solver developed by Niklas Eén and Niklas Sörensson, and described in more detail here: http://minisat.se. **MiniSat** is a very successful SAT solver: it has won several SAT competitions, and is widely used. It also has the unique feature of being very small, as a computer program: less than 4,000 lines of C++ in a recent version.

SAT solvers are generally easy to use, because they have fewer features than a BDD package. Roughly speaking: you give the solver a file specifying the clauses in CNF form, and the solver prints out some information about satisfiability: yes, it's SAT and here is an assignment to the inputs to satisfy it; no it is UNSAT. Also, the solver usually prints out something about the internal SAT solver process, e.g., aspects of the DPLL search, and the various tricks and strategies deployed to get an answer.

The most important thing to know about **MiniSat** is that it uses a famous, common text file format for its clause input: **DIMACS** format. Here is the basic format:

- A file can start with some comment lines. These are just text lines that start with a lower case "**c**". As an example:

```
c This is a comment line
c  … and this is too
```

- After the initial comments, the next line of the file must tell how many variables (**V** a positive integer) and how many clauses (**N** a positive integer) in this CNF format:

```
p cnf V N
```

- The next **N** lines of the file each specify one single clause. DIMACS format assumes your variables are **x1, x2, x3, …, xV**. You specify a *positive* literal (like **x2** or **x7**) in this clause with a positive integer (in this case, **2** or **7**). You

[1] **MiniSat** -- Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson. Copyright (c) 2007-2010 Niklas Sorensson

specify a *negative*, complemented literal wth a negative integer(so ¬**x5** is -**5**, and ¬**x23** is -**23**). You end the line with a **0.**

Example: Suppose we have this tiny CNF equation and we want to check it for SAT:

(x1 + ¬x3)(x2 + x3 + ¬x1)

Then the file in DIMACS format we need to input to **MiniSat** is this:

```
c Comment line begins by 'c'
c This is second comment line
p cnf 3 2
1 -3 0
2 3 -1 0
```

If we run **MiniSat**, it produces two sorts of outputs (1) some information sent to the command line about how the internal DPLL search process went; (2) a yes/no answer about satisfiability and any resulting satisfying assignment. So, if we run the above file, the "internal search information" returned is this:

```
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:            3                                          |
|  Number of clauses:              2                                          |
|  Parse time:                  0.00 s                                        |
|                                                                             |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
===============================================================================
restarts              : 1
conflicts             : 0              (0 /sec)
decisions             : 3              (0.00 % random) (3876 /sec)
propagations          : 3              (3876 /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 4.34 MB
CPU time              : 0.000774 s

SATISFIABLE
```

And the satisfiability answer returned is this:

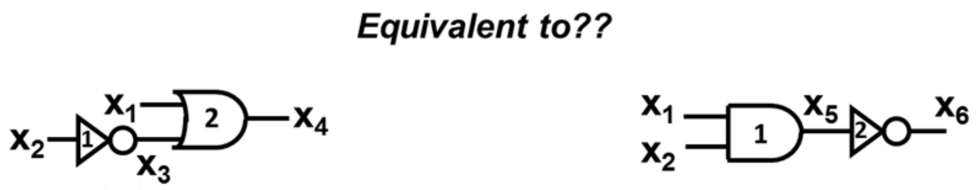
```
 SAT
-1 2 -3 0
```

The "`[ Problem Statistics ]`" output tells you some useful things: how many variables and clauses **MiniSat** found in your input file; how many variable decisions (assignments) and Boolean constraint propagations (BCP steps) it did; what sort of clause conflicts it found while doing DPLL search; and some other things related to the "clever" heuristics all modern SAT solvers use.

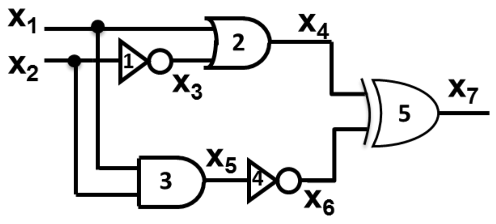
The satisfiability result tells you yes/or no if it was satisfiable: **Yes** in this case, since the result says **`SAT`**. And also an assignment of true/false values to the variables. If a number appear as a *positive integer* **k**, it means that variable **x_k=1** in the satisfying assignment. If a number appears as a *negative integer* **k**, it means that variable **x_k=0** in the satisfying assignment. So, in this example, **x1=0, x2=1, x3=0** is the satisfying assignment.

Realistic Usage Example

Lets try another, larger example, which is bit more realistic as a VLSI CAD problem: we want to compare these two very simple logic networks.



First, we connect the two circuits' outputs with an EXOR gate; then we use **MiniSat** to check if this new logic network can be satisfied **(x7=1)** or not. If it is satisfiable, we will know the original two circuits are not equivalent, because there exists one satisfying assignment such that **x4 = 1, x6=0** or **x4=0, x6=1**.



Here are the SAT clauses converted from this little logic network:

(x2+x3)(¬x2+¬x3)(¬x1+¬x2+x5) (x1+¬x5) (x2+¬x5)
(x1+x3+¬x4) (¬x1+x4) (¬x3+x4) (x5+x6)(¬x5+¬x6)
(x4+¬x6+x7) (x4+x6+¬x7) (¬x4+x6+x7) (¬x4+¬x6+¬x7)(x7)

Here is the DIMACS-format input file:

```
c There are 7 variables and 15 clauses
p cnf 7 15
2 3 0
-2 -3 0
-1 -2 5 0
1 -5 0
2 -5 0
1 3 -4 0
-1 4 0
-3 4 0
5 6 0
-5 -6 0
4 -6 7 0
4 6 -7 0
-4 6 7 0
-4 -6 -7 0
7 0
```

Here is the **MiniSat** output – first [Problem Statistics], then the satisfiability result:

```
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:             7                                         |
|  Number of clauses:              14                                         |
|  Parse time:                   0.00 s                                       |
|                                                                             |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
===============================================================================
restarts              : 1
conflicts             : 0              (0 /sec)
decisions             : 2              (0.00 % random) (2506 /sec)
propagations          : 7              (8772 /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 4.34 MB
CPU time              : 0.000798 s

SATISFIABLE

SAT
-1 2 -3 -4 -5 6 7 0
```

As we can see, the solver finds *one* satisfying assignment[2] It gives the assignment value for *every* variable in the input clauses. Since we only care about **x1** and **x2**, the "real" primary inputs to our logic, we can see that **x1=0** and **x2=1** will force the EXOR output =1, and this is the input we want: it makes the two logic networks

[2] Emphasis on the word "one" since there *could* be several solutions. When there is more than one solution, **MiniSAT** returns just one of them. Thus, the output that you get from running **MiniSat** might be different.

create *different* output values, and demonstrates that they are *not* the same Boolean function.

MiniSat Statistics Note

You will occasionally see something a bit strange in the **`[Problem Statistics]`** output: a place where you expect a number, but you see the word "**nan**". This is highlighted below:

```
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:             7                                         |
|  Number of clauses:              14                                         |
|  Parse time:                   0.00 s                                       |
|                                                                             |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
===============================================================================
restarts              : 1
conflicts             : 0              (0 /sec)
decisions             : 2              (0.00 % random) (2506 /sec)
propagations          : 7              (8772 /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 4.34 MB
CPU time              : 0.000798 s

SATISFIABLE
```

What is actually happening here is that the solver is doing some arithmetic and dividing something by 0. In the above example, this is because there are 0 conflict literals, and so an attempt to calculate some useful percentages results in a divide-by-0. Another common reason for the **nan** is because the **MiniSat** solver is using some system calls to measure the amount of CPU time each part of the computation takes. On some platforms, for very small problems, these computations are so fast that the system times returns a "0.0" result, for the number of time units (e.g., milliseconds). Then, when doing the calculations for "computations / second", the denominator is 0.0, and when you divide something by 0.0 – what happens? The answer on a modern computer with IEEE standard floating point (http://en.wikipedia.org/wiki/IEEE_floating-point_standard), is something called a **Not-a-Number**, which is abbreviated as **NaN** or **nan**. That is what is happening here. Divide something by 0.0, and this **nan** is the result.