181.144 Computer-Aided Verification

Project

Document version 1

15 May 2020

Overview

The goal of this project is to implement a Bounded Model Checker (BMC) for synchronous sequential circuits and simple safety properties. The project consists of two parts:

- Part 1: Implement a BMC that accepts a circuit as well as an integer k and checks that a given safety property holds for up to k steps.
- Part 2: Extend the BMC with interpolation in order to check the safety property for an unbounded number of steps.

This document is structured as follows: Sections 1 and 2 describe both parts of this project in more detail. Section 3 specifies constraints on your solution. Section 5 explains grading and submission modalities. Appendix A contains tutorials on model checking of synchronous sequential circuits, the AIGER file format and proof logging using MiniSat.

Please check the TUWEL course page [4] regularly, especially for official announcements or updated versions of this document. If you have any questions, especially regarding the task description, please get in touch via the TUWEL forum. For specific questions you can also drop us an email (`fweber@forsyte.at`, `ssalling@forsyte.at`).

1 First Part: Bounded Model Checker

In the first step, we ask you to implement a bounded model checker for synchronous circuits:

- As input, your BMC should accept a file in the ASCII variant of the AIGER format [3] as well as an integer k. The input file describes a synchronous sequential circuit in the form of an And-Inverter graph with latches. Appendix A.2 contains a tutorial on the AIGER format, and you

This project is heavily based on Georg Weissenbacher's project from 2019.

can find more information on the AIGER website [1]. Note that AIGER files can contain symbol tables and comments sections, your parser should ignore these.

- You can assume that the circuit has exactly one output bit which we also call the "bad state detector". Given an input circuit and the integer k, your model checker should verify that the safety property "output is always 0" holds for up to k steps. For an introduction to BMC of sequential circuits, refer to Appendix A.1. For an introduction to BMC in general, have a look at Sections 8.1-8.3 of the lecture notes [7] and the lecture slides.
- By design, BMC relies heavily on a SAT solver. We ask you to use MiniSat v1.14 [6], which accepts a file in the DIMACS CNF format [5]. A DIMACS CNF file contains a propositional formula in Conjunctive Normal Form. In order to pass arbitrary propositional formulas to MiniSat, you will need to implement a procedure that translates a propositional formula to an equisatisfiable formula in CNF. You can find some hints on this step in Section 8.3 in the lecture notes [7].
- Please provide a script called `run-part1` which expects two arguments (filename of AIGER file and an integer k), invokes your bounded model checker and prints `OK` if the given circuit satisfies the safety property up to k steps, and prints `FAIL` otherwise.

2 Second Part: BMC with Interpolation

The BMC from the first part will only be able to check the safety property for up until k steps, for a predefined k. It cannot prove that the safety property holds in general. In the second part of the project, we ask you to tackle this problem by extending your BMC from part 1 with *interpolation*.

- You can find an introduction to interpolation in Section 8.4.2 of the lecture notes [7].
- In order to extend your BMC with interpolation, you need to implement a procedure to compute interpolants from an unsatisfiable formula, e.g. the labelling algorithm from Section 8.4.2 of the lecture notes [7] which extracts an interpolant from a resolution proof tree. Please modify the proof-logging version of MiniSat v1.14 [6] such that it generates a resolution proof tree. One possible way to modify MiniSat accordingly is given in Appendix A.3, but feel free to integrate MiniSat into your BMC in a different way.
- Please provide a script called `run-part2` which expects one argument (filename of AIGER file), invokes your BMC extended with interpolation, and prints `OK` if the given circuit satisfies the safety property, and `FAIL` otherwise.

3 Constraints

Please note the following constraints:

- This course project is meant to be solved individually, group work is not permitted.
- We suggest you use C++, Python, Java or Scala for your implementation, but you are welcome to use any other (reasonable) programming language.
- We expect you to implement your BMC from scratch (but of course, you are allowed to use MiniSat). In particular, we expect you to write your own DIMACS generator and AIGER parser.

4 Benchmarks

You can find a collection of benchmarks on the AIGER website [1] in the file `tip-aig-20061215.zip` in the section "Sequential Model Checking AIGER Benchmarks". Each file specifies a circuit with a bad state detector. Note that you have to use the tool `aigtoaig` to convert binary AIGER files to the ASCII AIGER format.

As a minimal requirement for passing the first part, your solution is required to give correct answers for each of the following benchmarks for up to $k = 30$ within at most 10 minutes on an ordinary notebook. (Note that this is a very generous timeout, it should be possible to solve this way faster.)

- `texas.ifetch1^5.E`
- `vis.eisenberg.E`
- `texas.two_proc^1.E`

Your solution for the second part is required to give correct answers at least for the following benchmarks, again taking at most 10 minutes each:

- `nusmv.syncarb5^2.B`
- `vis.emodel.E`
- `cmu.gigamax.B`

For grading, your solution will also be tested on a random collection of different AIGER benchmarks.

5 Submission and Grading

Please upload your solution in TUWEL until **1st October 2020, 23:59 Vienna time**. Your solution must include:

- the documented source code of your BMC,
- the patched MiniSat source code, if you solved the second part,
- a README containing instructions how to build your BMC,
- everything we specified above, most notably the two scripts `run-part1` and `run-part2` to run your BMC.

After this deadline, we will conduct individual grading interviews in which you will be asked to compile and run your BMC on some examples, and to answer questions about your solution. Grading interviews will be individually scheduled. We would prefer the interviews to take place on campus, but depending on the situation in October, we might need to conduct the interviews online.

If you need your grade early (e.g. because you are about to graduate), please get in touch with us until end of June.

A solid solution of part 1 (plus a solid grading interview) are sufficient to get a passing grade for this Übung. In order to improve your grade, we strongly suggest to solve part 2 as well.

References

[1] *AIGER*. http://fmv.jku.at/aiger/.

[2] *And-inverter graphs (Wikipedia)*. https://en.wikipedia.org/wiki/And-inverter_graph.

[3] Armin Biere. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. http://fmv.jku.at/papers/Biere-FMV-TR-07-1.pdf.

[4] *Computer-Aided Verification (UE 2,0)*. https://tuwel.tuwien.ac.at/course/view.php?id=25117.

[5] *DIMACS CNF Files*. https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html.

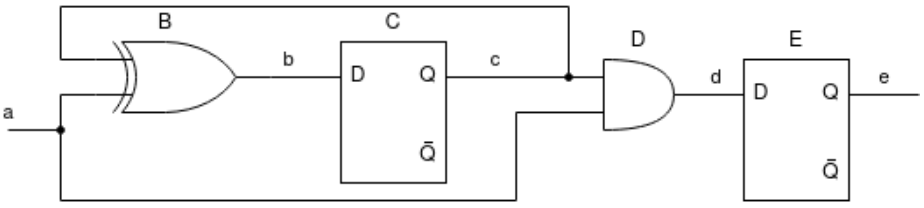
[6] *MiniSat*. http://minisat.se/MiniSat.html.

[7] Fabio Somenzi. *Computer-Aided Verification (Lecture Notes)*. https://tuwel.tuwien.ac.at/course/view.php?id=25120.

A Tutorials

A.1 BMC for synchronous sequential circuits

In this tutorial, we illustrate BMC of synchronous sequential circuits on a running example. Consider the following circuit:



The circuit has one input wire, one output wire and contains two gates – XOR gate B and AND gate D – and two latches C and E. As the circuit contains not only Boolean (combinatorial) gates, but also latches, we are dealing with a so-called sequential circuit. We can think of each latch as a storage unit for 1 bit of state. Sequential circuits have an implicit clock and latches update their values once per clock cycle. In each clock cycle, every wire of the circuit can be associated with a Boolean value (a bit). We have marked some wires in the circuit with symbols a to e.

In the following, clock cycles will be interpreted as *computation steps*. In each computation step, the circuit is provided with an input (wire a marked in the circuit). Hence, we can evaluate the combinatorial parts of the circuit once per step, with respect to the current input and the current values of the latches. This allows us to compute, in every step, the current output (e) as well as the new values of C and E, from the current input (a) and the current values of C (c) and E (e). Latches are initialized to zero.

In every step, the values of wires b and d can be expressed in terms of wires a and c:

$$b = c \oplus a \qquad \text{(note that we write } \oplus \text{ for XOR)}$$
$$d = c \wedge a$$

Let us simulate a possible sequence of steps:

1. Assume the first input bit a is 1. As latches are initialized to zero, $c = 0$, and consequently $b = (0 \oplus 1) = 1$. Hence, the new value of latch C will be 1. Likewise, $d = (0 \wedge 1) = 0$, so the new value of latch E is 0. As we initialize latches to 0, the output bit e is 0.

2. Assume the second input is $a = 0$. In the previous step, we stored 1 in latch C, so we have $c = 1$, and consequently $b = (1 \oplus 0) = 1$. Hence, the new value of latch C will be 1. Likewise, we can compute $d = (1 \wedge 0) = 0$, so the new value of latch E will be 0. As we stored 0 in latch E in the preceding step, we have $e = 0$ for the output bit.

We would now like to ensure that we will never reach a state in which the output bit e is 1, no matter which sequence of input bits we provide to the circuit. This is a safety property, "something bad ($e = 1$) will never happen", which we can formalize using LTL notation as $\mathsf{G}\neg e$.

Using proper terminology, we want to check if the circuit is a model for $\mathsf{G}\neg e$. For that, we use a bounded model checking technique: For a fixed number of steps k, we try to find a counterexample that falsifies the safety property in k steps. If this is not possible, the safety property holds for up to k steps. The task of finding a counterexample is delegated to a SAT solver: For a fixed k, we construct a propositional formula that is satisfiable iff. there is a counterexample that falsifies the safety property in k steps.

We first encode the behavior of the circuit using propositional formulas. We have already seen that each latch stores one bit of state, and we also consider the current input as part of the state. The circuit has one input and contains two latches C and E, so we use three bits c, e and a to describe the state of the circuit: c holds the value of latch C, and e holds the value of latch E, and a holds the current input.

Then, we express the *initial* states of the circuit as a propositional formula $I(c, e, a)$ (over three variables c, e and a) which is true if and only if c, e and a describe an initial state. This is the case precisely if c and e are both false, because latches are initialized to zero. Since the current input is not relevant, the variable a does not appear in the formula at all:

$$I(c, e, a) = \neg c \wedge \neg e$$

Next, we encode the transition relation of the circuit as a propositional formula $T(c_t, e_t, a_t, c_{t+1}, e_{t+1}, a_{t+1})$ over six variables: the current state (encoded as c_t, e_t and a_t) and the new state (encoded as c_{t+1}, e_{t+1} and a_{t+1}). We design the formula such that it is true if and only if the circuit is allowed to transition from the current state to the new state:

$$\begin{aligned} T(c_t, e_t, a_t, c_{t+1}, e_{t+1}, a_{t+1}) &= (c_{t+1} \leftrightarrow b_t) \wedge (e_{t+1} \leftrightarrow d_t) \\ &= (c_{t+1} \leftrightarrow (c_t \oplus a_t)) \wedge (e_{t+1} \leftrightarrow (a_t \wedge c_t)) \end{aligned}$$

As the new input does not influence the decision whether a transition is valid or not, the symbol a_{t+1} does not appear in the formula. We can convince ourselves that this definition matches our intuition for some examples:

- $T(0, 0, 0, 1, 0, 0)$ evaluates to 0. Indeed, if latch C stores 0 and the input bit is 0, there is no way the new value of latch C can be 1.
- $T(0, 1, 0, 0, 0, 1)$ evaluates to 1, and the corresponding transition is legal: If latch C currently stores 0, and E stores 1, an input of 0 will keep C at 0 and flip E to 0.

We now combine these formulas to simulate the circuit for a fixed number of steps k. For example, for $k = 2$, we obtain the propositional formula

$$I(c_0, e_0, a_0) \wedge T(c_0, e_0, a_0, c_1, e_1, a_1) \wedge T(c_1, e_1, a_1, c_2, e_2, a_2)$$

or, if we expand the definitions of I and T:

$$\begin{aligned}
&\underbrace{\neg c_0 \wedge \neg e_0}_{I(c_0,e_0,a_0)} \\
\wedge\ &\underbrace{(c_1 \leftrightarrow c_0 \oplus a_0) \wedge (e_1 \leftrightarrow (a_0 \wedge c_0))}_{T(c_0,e_0,a_0,c_1,e_1,a_1)} \\
\wedge\ &\underbrace{(c_2 \leftrightarrow c_1 \oplus a_1) \wedge (e_2 \leftrightarrow (a_1 \wedge c_1))}_{T(c_1,e_1,a_1,c_2,e_2,a_2)}
\end{aligned}$$

Every satisfying variable assignment of this formula describes a legal sequence of two steps (e.g. the assignment given by $c_0 = e_0 = 0$, $a_0 = 1$, $c_1 = 1$, $e_1 = 0$, $a_1 = 0$, $c_2 = 1$ and $e_2 = 0$ captures the two steps we simulated earlier).

Recall that we do not just want to simulate the circuit, but want to check a safety property. For a fixed k, we construct a formula $[\![\neg \mathsf{G} \neg e]\!]_k$ that is true iff. the property is falsified in k steps — in other words, if we can reach a counterexample in k steps. A counterexample to $\mathsf{G}\neg e$ is a state in which e is 1. Hence, we set

$$[\![\neg \mathsf{G} \neg e]\!]_k = e_0 \vee e_1 ... \vee e_k$$

For a fixed k, we can now combine formulas to obtain a formula that is satisfiable iff. it is possible to provide inputs to the circuit such that $\mathsf{G}\neg e$ is falsified in k steps. First consider $k = 0$, where no transitions is made. We construct the formula:

$$I(c_0, e_0, a_0) \wedge [\![\neg \mathsf{G} \neg e]\!]_0 = \neg c_0 \wedge \neg e_0 \wedge e_0$$

which is unsatisfiable. Consequently, the initial states of the circuit satisfy the safety property.

We continue with $k = 1$ and obtain a more complicated formula:

$$\underbrace{\neg c_0 \wedge \neg e_0}_{I(c_0,e_0,a_0)} \wedge \underbrace{(c_1 \leftrightarrow (c_0 \oplus a_0)) \wedge (e_1 \leftrightarrow (a_0 \wedge c_0))}_{T(c_0,e_0,a_0,c_1,e_1,a_1)} \wedge \underbrace{(e_0 \vee e_1)}_{[\![\neg G \neg e]\!]_1}$$

If we translate this formula to CNF, we can pass it to a SAT solver. A SAT solver will tell us that this formula is, in fact, unsatisfiable. Hence, we now know that the circuit satisfies the safety property if we perform one step.

As a last example, we construct the formula for $k = 2$:

$$\begin{aligned}
&\underbrace{\neg c_0 \wedge \neg e_0}_{I(c_0,e_0,a_0)} \\
\wedge\ &\underbrace{(c_1 \leftrightarrow c_0 \oplus a_0) \wedge (e_1 \leftrightarrow (a_0 \wedge c_0))}_{T(c_0,e_0,a_0,c_1,e_1,a_1)} \\
\wedge\ &\underbrace{(c_2 \leftrightarrow c_1 \oplus a_1) \wedge (e_2 \leftrightarrow (a_1 \wedge c_1))}_{T(c_1,e_1,a_1,c_2,e_2,a_2)} \\
\wedge\ &\underbrace{(e_0 \vee e_1 \vee e_2)}_{[\![\neg G \neg e]\!]_2}
\end{aligned}$$

As it turns out, this formula is satisfiable — a SAT solver will be able to give us a satisfying assignment, for example $c_0 = e_0 = 0$, $a_0 = 1$, $c_1 = 1$, $e_1 = 0$, $a_1 = 1$, $c_2 = 0$, $e_2 = 1$. Hence, the safety property is violated! If we provide the input sequence $(a_0, a_1) = (1, 1)$, the output bit is set to 1.

In this tutorial, we have seen how we can translate k computation steps of the given sequential circuit to a propositional formula that is satisfiable iff. the safety property "output is always 0" is violated in k steps. A more thorough introduction to BMC can be found in Sections 8.1 to 8.3 of the lecture notes [7].

A.2 AIGER

In this section, we introduce the AIGER file format [1, 3], which can be used to specify synchronous sequential circuits, or more specifically, And-Inverter graphs [2] with latches. The AIGER format comes in a binary and an ASCII variant, but we will only consider the ASCII variant.

The header, i.e., the first line, of an AIGER file specifies the number of inputs, outputs, latches and AND gates in the circuit, as well as the number of variables which are used to specify the behavior of the circuit. It has the following shape:

$$\texttt{aag}\ M\ I\ L\ O\ A$$

where M, I, L, O, A are integers with the following interpretation:

- M is the maximum variable index, e.g. if $M = 5$, we can have variables x_1, x_2, ..., x_5.
- I is the number of input wires, or input bits.
- L is the number of latches in the circuit. Each latch stores 1 bit of state.
- O is the number of output wires, or output bits.
- A is the number of AND gates in the circuit.

The rest of the lines in the file refer to the variables $x_1, ..., x_M$ to specify the behavior of the circuit. Literals, i.e., variables $x_1, ..., x_M$ and their negations, as well as boolean constants are encoded as integers. An integer n encodes the following:

- If $n = 0$, it refers to the constant false (the constant 0)
- If $n = 1$, it refers to the constant true (the constant 1)
- If $n \geq 2$ is even, it refers to literal x_i with $i = n/2$
- If $n \geq 2$ is odd, it refers to literal $\neg x_i$ with $i = (n-1)/2$

For the remainder of this section, we will use literals x_1, $\neg x_1$, ... and their integer encodings interchangeably. The following table illustrates the encoding:

n	0	1	2	3	4	5	6	...
term	false	true	x_1	$\neg x_1$	x_2	$\neg x_2$	x_3	...

Now that we know how variables are referenced, let us have a look at the lines following the header. Note that the header tells us exactly how many subsequent lines to expect: First, there are I lines describing the circuit inputs, then L lines for the latches, O lines for the outputs and finally A lines for the

AND gates. Note that the AIGER file might additionally contain a symbol table, which is specified by lines starting with the characters `i`, `o` or `l`, or a comments section whose beginning is marked with the character `c` (see [3] for more information). Symbol tables as well as comment sections can be safely ignored.

The lines after the header specify the circuit as follows:

- For each input wire (from 1 to I), we have one line containing a literal which will be associated with the respective input. For example the lines

  ```
  2
  6
  ```

  state that the variable x_1 is set to the first input (first line), and that the variable x_3 corresponds to the second input (second line). You can assume that only positive literals, i.e., even numbers, are given here.

- For each latch (from 1 to L), we have one line containing two literals Q and Q' separated by a space. Q refers to a variable that will hold the current value of the latch in each step. You can assume that Q is always even. Q' corresponds to the literal which determines the new value of the latch in each step. For example the lines

  ```
  6 8
  4 5
  ```

  specify two latches: The variable x_3 will hold the current value of the first latch, and the new value of the latch is given by the literal x_4. The current value of the second latch is given by the variable x_2, and the new value of the second latch is given by $\neg x_2$, i.e., the latch flips its value in each step. Latches are initialized to zero.

- For each output wire (from 1 to O), we have one line containing a literal which determines the value of the output wire. For example the lines

  ```
  3
  1
  ```

  specify that the first output bit is $\neg x_1$, and the second output bit is constant 1.

- For each AND gate (from 1 to A), we have one line containing three literals:

  $$o \;\; i_1 \;\; i_2$$

  which defines the value of the variable corresponding to o to be the conjunction of the literals identified by i_1 and i_2. You can assume that o is

always even. For example the lines

```
4 7 8
6 10 3
```

define two AND gates: The first AND gate sets the variable x_2 to $\neg x_3 \wedge x_4$. The second AND gate sets the variable x_3 to $x_5 \wedge \neg x_1$.

Let us put everything together and look at some annotated examples (with line numbers). The following circuit from the AIGER technical report [3] builds an inverter (a NOT gate):

1	`aag 1 1 0 1 0`	one variable x_1, one input bit, no latches, one output bit, no AND gates
2	`2`	set x_1 to the first (and only) input
3	`3`	set the first (and only) output to $\neg x_1$

The following circuit, also taken from [3], builds an OR gate:

1	`aag 3 2 0 1 1`	three variables x_1, x_2, x_3, two inputs, no latches, one output, one AND gate
2	`2`	set x_1 to the first input
3	`4`	set x_2 to the second input
4	`7`	set the output to $\neg x_3$
5	`6 3 5`	set x_3 to $\neg x_1 \wedge \neg x_2$

Note that the output of the circuit is exactly $\neg(\neg x_1 \wedge \neg x_2)$, which is equivalent to $x_1 \vee x_2$.

Consider an example with latches, also from [3]:

1	`aag 1 0 1 2 0`	one variable x_1, no inputs, one latch, two outputs, no AND gates
2	`2 3`	set x_1 to the current value of the latch, its new value will be determined by $\neg x_1$
3	`2`	set the first output to x_1
4	`3`	set the second output to $\neg x_1$

This file specifies a circuit without inputs, whose outputs will flip every step. As latches are initialized to zero, the first step will set the first output to 0 and the second output to 1. The new value of the latch will be 1, so the next step will set the outputs to 1 and 0, respectively.

Finally, let us take a look at a more complex example:

1	`aag 6 1 2 1 3`	six variables x_1, ..., x_6, one input bit, two latches, one output bit, and three AND gates
2	`2`	set x_1 to be the first (and only) input
3	`4 12`	set x_2 to current value of the first latch, its new value will be determined by variable x_6
4	`10 6`	set x_5 to current value of the second latch, its new value will be determined by variable x_3
5	`10`	set the first (and only) output to x_5
6	`6 2 4`	set $x_3 = x_1 \wedge x_2$
7	`8 3 5`	set $x_4 = \neg x_1 \wedge \neg x_2$
8	`12 7 9`	set $x_6 = \neg x_3 \wedge \neg x_4$

This might look complicated at first, but as it turns out, the file exactly describes the circuit from Section A.1: According to line 2, x_1 holds the current input, which we call a in the circuit. Line 3 now defines a latch that corresponds to latch C, and states that variable x_2 holds the current value of latch C (we called this value c). Hence, line 6 sets $x_3 = a \wedge c$. Line 7 then sets $x_4 = \neg a \wedge \neg c$. If we substitute both in line 8, we get $x_6 = \neg(a \wedge c) \wedge \neg(\neg a \wedge \neg c)$, which we can transform to $x_6 = (\neg a \vee \neg c) \wedge (a \vee c)$ — this corresponds to $a \oplus c$. Line 3 states that the new value of latch C is determined by x_6, so in each step, we calculate the new value of C to be $a \oplus c$. Line 4 defines a latch that corresponds to latch E: x_5 represents its current value (we called this e) and its new value is determined by x_3, which we have seen to be equivalent to $a \wedge c$. Finally, line 5 states that the output is set to x_5, the current value of latch E.

A.3 Proof Logging in MiniSat

In order to solve the second part of the project, you need to implement a procedure that constructs an interpolant from a unsatisfiable formula. Section 8.4.2 of the lecture notes [7] describes an algorithm for extracting an interpolant from a resolution proof tree. See Section 7.1 of the lecture notes for an introduction to resolution.

In the following, we give some hints how to modify MiniSat such that it generates a proof tree for unsatisfiable formulas. Note that the following describes only one possible solution, and you are very welcome to try other, more sophisticated ideas.

First, download the proof-logging version of MiniSat v1.14 from the MiniSat page [6] (`MiniSat-p_v1.14.src.zip`). In the source file `Main.c`, you will find the following code section, starting with line 224:

```
224  struct Checker : public ProofTraverser {
225    vec<vec<Lit> > clauses;
226
227    void root (const vec<Lit>& c) {
228      //**/printf("%d: ROOT", clauses.size()); for (int i = 0; i < c.size(); i++) printf(" %
           s%d", sign(c[i])?"-":"", var(c[i])+1); printf("\n");
229      clauses.push();
230      c.copyTo(clauses.last()); }
231
232    void chain (const vec<ClauseId>& cs, const vec<Var>& xs) {
233      //**/printf("%d: CHAIN %d", clauses.size(), cs[0]); for (int i = 0; i < xs.size(); i++)
             printf(" [%d] %d", xs[i]+1, cs[i+1]);
234      clauses.push();
235      vec<Lit>& c = clauses.last();
236      clauses[cs[0]].copyTo(c);
237      for (int i = 0; i < xs.size(); i++)
238        resolve(c, clauses[cs[i+1]], xs[i]);
239      //**/printf(" =>"); for (int i = 0; i < c.size(); i++) printf(" %s%d", sign(c[i])
             ?"-":"", var(c[i])+1); printf("\n");
240      }
241
242    void deleted(ClauseId c) {
243      clauses[c].clear(); }
244  };
```

If you uncomment lines 228, 233 and 239 and recompile MiniSat, the new MiniSat binary will print a proof tree if the command-line option `-c` is passed. The proof tree is given as a sequence of numbered lines. Each line assigns a number to a clause that was either taken from the input DIMACS CNF file, or obtained by repeatedly applying the resolution rule to existing clauses.

A line like

12: ROOT −13 −25

denotes that the clause 12 consists of literals −13 −25 and was taken from the input CNF. A line like

133: CHAIN 105 [12] 127 [6] 126 [11] 90 [5] 85 => 1 2 3 4 7 8 9 10 −17

describes that the clause 133 contains literals 1 2 3 4 7 8 9 10 −17 and was obtained by applying a chain of resolution steps as follows:

1. Clause 105 and 127 were resolved on literal 12
2. this resolvent and clause 126 were resolved on literal 6
3. this resolvent and clause 90 were resolved on literal 11
4. this resolvent and clause 85 were resolved on literal 5, finally producing clause 133

A resolution proof is a derivation of the empty clause from the input CNF. Hence, the proof ends with a line like

1121: CHAIN 1117 [30] 1120 =>

describing that clause 1121, obtained by resolving clauses 1117 and 1120 on literal 30, is empty.