Appendix A

The Prover for Synchronous Systems with No Learning

A.1 Availability

The theorem prover for $\mathsf{KL}^{sync,nl}_{(n)}$) can be downloaded from

`http://www.csc.liv.ac.uk/~claudia/.`

It requires Otter (version 2.2), SWI-Prolog (version 4.0.8) and emacs (version 19). There are newer versions, but these are the oldest versions of Otter, Prolog, and emacs with which the theorem prover has been tested. Fortunately, these are all free software and easily available:

- **Emacs:** `http://ftp.gnu.org/pub/gnu/emacs/`
- **Otter/Mace:** `http://www-unix.mcs.anl.gov/AR/otter/`
- **SWI-Prolog:** `http://www.swi-prolog.org/`

Standard installation of Emacs and Prolog (for Linux/Unix systems) usually places the executable in `\usr\bin\` (or `\usr\local\bin\`, for local installation).

Otter (and Mace, a software for generating models and counter-examples), however, installs in `\usr\local\otter-3.3\bin`. Therefore, make sure that this directory is in your home path. The same applies for the other two programs, if you usually opt for non-standard installations.

A.2 Usage

After downloading the file, uncompress it by typing

```
tar -xzvf klsnl.tar.gz
```

at the command line in the directory the file has been downloaded. Move into the prover directory:

```
cd klsnl
```

and start emacs:

```
emacs &
```

If you are using a shell other than bash, modify the file tl-otter.el at line 278 from "/bin/bash" to "bin/your-shell". Load this file by typing

```
[ESC X] load-file [RETURN] tl-otter.el [RETURN].
```

Invoke the shell by typing

```
[ESC X] shell [RETURN].
```

In the shell buffer, start Prolog by typing

```
pl -s loadtlk.pl -L0 -G0 -T0 [RETURN].
```

The argument "-s" loads the file loadtlk.pl. This file loads all Prolog files that will be used by the prover. The other arguments, i.e. "-L0 -G0 -T0", instructs Prolog to use the maximum memory for local, global, and trail stacks, respectively. In order to run a example, type

```
otres([List]).
```

at the Prolog prompt. List should be a set of well-formed formulae in the language of the prover, where symbols from $\mathsf{KL}_{(1)}^{snl}$ are translated as follows:

Operators										
Propositional				Knowledge	Temporal					
$\neg$	$\vee$	$\wedge$	$\Rightarrow$	K_1	$\bigcirc$	$\Box$	$\Diamond$	$\mathcal{U}$	$\mathcal{W}$
neg	or	and	imp	k	next	always	sometime	until	unless

Table A.1: Logical symbols in the language of the prover.

Propositional symbols are sequences of letters or numbers starting with a lower-case letter[1]. Terms starting by `tmpp` followed by a number, and the functors `new()` and `nkn()`, are regarded as reserved symbols and should not be used by the user as a name for a proposition. Note that as we are restricted to the single-agent case the knowledge operator is written as "k", i.e. we drop the index of the knowledge operator. Precedence of operators is as usual and parentheses can be dropped where there is no ambiguous reading. Make sure to write a space between operators and/or between operators and propositional symbols. Otherwise, the parser will interpret, for instance, "kp" as a propositional symbol, instead of $\mathrm{K}\, p$.

If you want to prove a formula is valid, first negate it. If the prover terminates with `Formula is unsatisfiable`, then the original formula is valid. For instance, to prove that the formula $\mathrm{K} \bigcirc p \Rightarrow \bigcirc \mathrm{K}\, p$ is valid, run

[1] The parser is not fully implemented, so the Prolog modules would accept any symbol that is written accordingly with the usual syntax for Prolog terms. However, the lisp interpreter would not deal with these terms properly. If you enter a name with a different syntax for a proposition, you will probably have to start everything from the very beginning again.

```
otres([neg(k next p imp next k p)]).
```

at the Prolog prompt. The prover will translate the formula into SNF_{snl} and do modal resolution. Once modal resolution is finished, i.e. no more literal or modal clauses have been generated, the program will show a modified prompt

```
|:
```

after listing the current set of literal and modal clauses. This prompt means that the Prolog will be waiting for the results returning from Otter. Otter is called at this point by the emacs lisp interface. In order to invoke the lisp function that will read the Usable and SoS clauses, run Otter, and translate back the result to Prolog, type

```
[ESC X]tlo-run-otter[RETURN].
```

Repeat this command every time the modified prompt is shown. The results generated by Otter are also kept in separate files which will be written in the directory the prover is. Each file is assigned an unique name, starting by `otter-output`. Those files are kept for two reasons: to have access to Otter's statistics and because the lisp interface does not translate to Prolog the justification for clauses generated by Otter. In case, you want to read either statistics or justifications, you have to locate the relevant `otter-output` file and open it. The easiest way of locating these files is by listing the `otter-output` files by time, that is, type `ls -l otter-output* --sort time` at another shell. You have to keep track of the time that Otter was running, but as it is now, there is no other way of looking at Otter's result. The proof proceeds until false has been generated or no new clauses have been generated.

A.3 Output

We explain in this section the meaning of the messages that comprise the output of the prover for $\mathsf{KL}_{(1)}^{sync,nl}$. The prover runs in a very verbose mode. If you want to run it more silently, you can change the first lines of the file `output.pl` from

```
/* Remove comments if you do not wish a very verbose output */

% my_writef(_X,_Y):-true,!.
% my_write(_X):-true,!.
% write_ruleset(_X):-true,!.

/* For very verbose output */

 my_writef(X,Y):-writef(X,Y).
 my_write(X):-write(X).
```

to

```
/* Remove comments if you do not wish a very verbose output */

 my_writef(_X,_Y):-true,!.
 my_write(_X):-true,!.
 write_ruleset(_X):-true,!.

/* For very verbose output */

% my_writef(X,Y):-writef(X,Y).
% my_write(X):-write(X).
```

By removing the comments as shown above, the prover will only provide as output lines indicating which modal/literal cycle is being performed, the loops found during temporal resolution, the clauses to be translated by the lisp interface to Otter, the result and statistics. We start to explain the meaning of the output by giving the full output of example `ex354`, commenting it when required. After loading the files `loadtlk.pl` and `examples.pl`, and typing `ex354.` at the prompt line, the prover shows the translation into SNF_{snl}, starting by showing the sometime clauses:

```
The list of eventuality rules is


[
  3 [] x => <>p
]
```

Then it shows that the prover is generating the SNL clauses:

```
Generating SNL Clauses


Temporal Clauses are
[
  5 [] x => Oy
  8 [] t => O~p
  9 [] t => Ot
]


New Temporal Clauses are
[
  5 [] x => Oy
```

```
  10 [5] nkn(x) => Onkn(y)
  8 [] t => O~p
  15 [8] nkn(t) => Onkn(neg p)
  9 [] t => Ot
  20 [9] nkn(t) => Onkn(t)
]
```

And, finally gives the full set of step, literal, and modal clauses:

```
Set of clauses after translation to SNF :


Initial Clauses
[
  1 [] tmpp1
]


Literal Clauses
[
  2 [] (~tmpp1 | x)
  4 [] (~x | ~p)
  6 [] (~y | ~p)
  7 [] (~y | t)
]


Modal Clauses
[
  21 [9] nkn(t) => ~k~t
  22 [9] ~nkn(t) => k~t
  23 [9] nkn(t) => ~k~t
```

```
  24 [9] ~nkn(t) => k~t
  16 [8] nkn(t) => ~k~t
  17 [8] ~nkn(t) => k~t
  18 [8] nkn(neg p) => ~kp
  19 [8] ~nkn(neg p) => kp
  11 [5] nkn(x) => ~k~x
  12 [5] ~nkn(x) => k~x
  13 [5] nkn(y) => ~k~y
  14 [5] ~nkn(y) => k~y
]

Temporal Clauses
[
  5 [] x => Oy
  10 [5] nkn(x) => Onkn(y)
  8 [] t => O~p
  15 [8] nkn(t) => Onkn(neg p)
  9 [] t => Ot
  20 [9] nkn(t) => Onkn(t)
]
```

The clauses are preceded by their numbers and by a list of numbers. The list of numbers is empty if the clause is in the initial set of clauses or if it was generated by Otter. If the list is not empty, the numbers in the list give the justification for a generated clause. In the above, for instance, clause number 10 was generated from clause number 5.

After translating and numbering the clauses, literal and modal resolution take place:

```
Cycle Number1.

Old Literal Clauses
[
]

Old Modal Clauses
[
]

New Literal Clauses
[
  27 [17, mres5] (nkn(t) | ~t)
  28 [19, mres5] (nkn(neg p) | p)
  29 [12, mres5] (nkn(x) | ~x)
  30 [14, mres5] (nkn(y) | ~y)
  2 [] (~tmpp1 | x)
  4 [] (~x | ~p)
  6 [] (~y | ~p)
  7 [] (~y | t)
]

New Modal Clauses
[
  16 [8] nkn(t) => ~k~t
  17 [8] ~nkn(t) => k~t
  18 [8] nkn(neg p) => ~kp
  19 [8] ~nkn(neg p) => kp
```

```
  11 [5] nkn(x) => ~k~x
  12 [5] ~nkn(x) => k~x
  13 [5] nkn(y) => ~k~y
  14 [5] ~nkn(y) => k~y
]
```

where the `Old Literal Clauses` and `Old Modal Clauses` are empty sets, because this is the first time that resolution is being performed. Then, once resolution takes place, the clauses in `New` are moved to `Old`, and the result of applying resolution between the clauses in those sets are given as `New Literal Clauses` and `New Modal Clauses`:

```
Cycle Number2.
Old Literal Clauses
[
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  29 [12, mres5] (nkn(x) | ~x)
  28 [19, mres5] (nkn(neg p) | p)
  27 [17, mres5] (nkn(t) | ~t)
]

Old Modal Clauses
[
  14 [5] ~nkn(y) => k~y
  13 [5] nkn(y) => ~k~y
```

```
  12 [5] ~nkn(x) => k~x
  11 [5] nkn(x) => ~k~x
  19 [8] ~nkn(neg p) => kp
  18 [8] nkn(neg p) => ~kp
  17 [8] ~nkn(t) => k~t
  16 [8] nkn(t) => ~k~t
]

New Literal Clauses
[
  33 [4, 2] (~p | ~tmpp1)
  38 [17, 7] (nkn(t) | ~y)
  42 [19, 6] (nkn(neg p) | ~y)
  41 [19, 4] (nkn(neg p) | ~x)
  46 [12, 2] (nkn(x) | ~tmpp1)
]

New Modal Clauses
[
  36 [16, 27] t => ~k~t
  39 [18, 28] ~p => ~kp
  43 [11, 29] x => ~k~x
  44 [11, 4] nkn(x) => ~kp
  47 [13, 30] y => ~k~y
  48 [13, 6] nkn(y) => ~kp
  49 [13, 7] nkn(y) => ~k~t
]
```

The prover continues performing literal/modal resolution until no new clauses are generated:

```
Cycle Number3.
Old Literal Clauses
[
  46 [12, 2] (nkn(x) | ~tmpp1)
  41 [19, 4] (nkn(neg p) | ~x)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  29 [12, mres5] (nkn(x) | ~x)
  28 [19, mres5] (nkn(neg p) | p)
  27 [17, mres5] (nkn(t) | ~t)
]

Old Modal Clauses
[
  49 [13, 7] nkn(y) => ~k~t
  48 [13, 6] nkn(y) => ~kp
  47 [13, 30] y => ~k~y
  44 [11, 4] nkn(x) => ~kp
  43 [11, 29] x => ~k~x
```

```
  39 [18, 28] ~p => ~kp
  36 [16, 27] t => ~k~t
  14 [5] ~nkn(y) => k~y
  13 [5] nkn(y) => ~k~y
  12 [5] ~nkn(x) => k~x
  11 [5] nkn(x) => ~k~x
  19 [8] ~nkn(neg p) => kp
  18 [8] nkn(neg p) => ~kp
  17 [8] ~nkn(t) => k~t
  16 [8] nkn(t) => ~k~t
]

New Literal Clauses
[
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
  104 [49, 14] (nkn(t) | ~nkn(y))
]

New Modal Clauses
[
  74 [43, 2] tmpp1 => ~k~x
  85 [44, 29] x => ~kp
  86 [44, 46] tmpp1 => ~kp
  102 [48, 30] y => ~kp
  111 [49, 30] y => ~k~t
```

```
]

Cycle Number4.
Old Literal Clauses
[
  104 [49, 14] (nkn(t) | ~nkn(y))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  46 [12, 2] (nkn(x) | ~tmpp1)
  41 [19, 4] (nkn(neg p) | ~x)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  29 [12, mres5] (nkn(x) | ~x)
  28 [19, mres5] (nkn(neg p) | p)
  27 [17, mres5] (nkn(t) | ~t)
]

Old Modal Clauses
[
  111 [49, 30] y => ~k~t
```

```
  102 [48, 30] y => ~kp
  86 [44, 46] tmpp1 => ~kp
  85 [44, 29] x => ~kp
  74 [43, 2] tmpp1 => ~k~x
  49 [13, 7] nkn(y) => ~k~t
  48 [13, 6] nkn(y) => ~kp
  47 [13, 30] y => ~k~y
  44 [11, 4] nkn(x) => ~kp
  43 [11, 29] x => ~k~x
  39 [18, 28] ~p => ~kp
  36 [16, 27] t => ~k~t
  14 [5] ~nkn(y) => k~y
  13 [5] nkn(y) => ~k~y
  12 [5] ~nkn(x) => k~x
  11 [5] nkn(x) => ~k~x
  19 [8] ~nkn(neg p) => kp
  18 [8] nkn(neg p) => ~kp
  17 [8] ~nkn(t) => k~t
  16 [8] nkn(t) => ~k~t
]

New Literal Clauses
[
]

New Modal Clauses
[
```

```
  120 [78, 12] ~nkn(neg p) => k~x
  124 [95, 14] ~nkn(neg p) => k~y
  127 [104, 14] ~nkn(t) => k~y
]

Cycle Number5.
Old Literal Clauses
[
  104 [49, 14] (nkn(t) | ~nkn(y))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  46 [12, 2] (nkn(x) | ~tmpp1)
  41 [19, 4] (nkn(neg p) | ~x)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  29 [12, mres5] (nkn(x) | ~x)
  28 [19, mres5] (nkn(neg p) | p)
  27 [17, mres5] (nkn(t) | ~t)
]
```

```
Old Modal Clauses
[
  127 [104, 14] ~nkn(t) => k~y
  124 [95, 14] ~nkn(neg p) => k~y
  120 [78, 12] ~nkn(neg p) => k~x
  111 [49, 30] y => ~k~t
  102 [48, 30] y => ~kp
  86 [44, 46] tmpp1 => ~kp
  85 [44, 29] x => ~kp
  74 [43, 2] tmpp1 => ~k~x
  49 [13, 7] nkn(y) => ~k~t
  48 [13, 6] nkn(y) => ~kp
  47 [13, 30] y => ~k~y
  44 [11, 4] nkn(x) => ~kp
  43 [11, 29] x => ~k~x
  39 [18, 28] ~p => ~kp
  36 [16, 27] t => ~k~t
  14 [5] ~nkn(y) => k~y
  13 [5] nkn(y) => ~k~y
  12 [5] ~nkn(x) => k~x
  11 [5] nkn(x) => ~k~x
  19 [8] ~nkn(neg p) => kp
  18 [8] nkn(neg p) => ~kp
  17 [8] ~nkn(t) => k~t
  16 [8] nkn(t) => ~k~t
]
```

```
New Literal Clauses
[
]

New Modal Clauses
[
]
```

After literal/modal resolution is completed, the prover shows the current set of literal and modal clauses:

```
Have generated no new resolvents,
current literal/modal clauses are :-
[
  104 [49, 14] (nkn(t) | ~nkn(y))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  46 [12, 2] (nkn(x) | ~tmpp1)
  41 [19, 4] (nkn(neg p) | ~x)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  29 [12, mres5] (nkn(x) | ~x)
```

```
  28 [19, mres5] (nkn(neg p) | p)
  27 [17, mres5] (nkn(t) | ~t)
]
[
  127 [104, 14] ~nkn(t) => k~y
  124 [95, 14] ~nkn(neg p) => k~y
  120 [78, 12] ~nkn(neg p) => k~x
  111 [49, 30] y => ~k~t
  102 [48, 30] y => ~kp
  86 [44, 46] tmpp1 => ~kp
  85 [44, 29] x => ~kp
  74 [43, 2] tmpp1 => ~k~x
  49 [13, 7] nkn(y) => ~k~t
  48 [13, 6] nkn(y) => ~kp
  47 [13, 30] y => ~k~y
  44 [11, 4] nkn(x) => ~kp
  43 [11, 29] x => ~k~x
  39 [18, 28] ~p => ~kp
  36 [16, 27] t => ~k~t
  14 [5] ~nkn(y) => k~y
  13 [5] nkn(y) => ~k~y
  12 [5] ~nkn(x) => k~x
  11 [5] nkn(x) => ~k~x
  19 [8] ~nkn(neg p) => kp
  18 [8] nkn(neg p) => ~kp
  17 [8] ~nkn(t) => k~t
  16 [8] nkn(t) => ~k~t
```

```
]
```

followed by the translation of literal clauses (as the usable set) and the temporal clauses (as the set of support) in the language of Otter:

```
otres_allres (1) ==> Usable rules are
s_nkn_t | -s_nkn_y.
s_nkn_neg_p | -s_nkn_y.
s_nkn_neg_p | -s_nkn_x.
s_nkn_neg_p | -s_tmpp1.
s_nkn_x | -s_tmpp1.
s_nkn_neg_p | -s_x.
s_nkn_neg_p | -s_y.
s_nkn_t | -s_y.
-s_p | -s_tmpp1.
-s_y | s_t.
-s_y | -s_p.
-s_x | -s_p.
-s_tmpp1 | s_x.
s_nkn_y | -s_y.
s_nkn_x | -s_x.
s_nkn_neg_p | s_p.
s_nkn_t | -s_t.


otres_allres (1) ==> Rules in the SoS are
-slast_x | s_y.
-slast_nkn_x | s_nkn_y.
-slast_t | -s_p.
```

```
-slast_nkn_t | s_nkn_neg_p.

-slast_t | s_t.

-slast_nkn_t | s_nkn_t.
```

After invoking the command `tlo-run-otter` at the modified prompt, the prover reads and shows the output coming from Otter.

```
Output from otter ==> New literal rules (A imp next false) are
[
]


Output from otter ==> Remaining rules (after snl) are
[
  104 [49, 14] (nkn(t) | ~nkn(y))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  46 [12, 2] (nkn(x) | ~tmpp1)
  41 [19, 4] (nkn(neg p) | ~x)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  29 [12, mres5] (nkn(x) | ~x)
  28 [19, mres5] (nkn(neg p) | p)
```

```
27 [17, mres5] (nkn(t) | ~t)

5 [] x => Oy

218 [5] nkn(x) => Onkn(y)

10 [5] nkn(x) => Onkn(y)

223 [10] nkn(x) => Onkn(y)

8 [] t => O~p

224 [8] nkn(t) => Onkn(neg p)

15 [8] nkn(t) => Onkn(neg p)

229 [15] nkn(t) => Onkn(neg p)

9 [] t => Ot

230 [9] nkn(t) => Onkn(t)

20 [9] nkn(t) => Onkn(t)

235 [20] nkn(t) => Onkn(t)

209 [] x => Onkn(y)

236 [209] nkn(x) => Onkn(y)

210 [] x => O~p

239 [210] nkn(x) => Onkn(neg p)

211 [] x => Ot

244 [211] nkn(x) => Onkn(t)

212 [] x => Onkn(t)

249 [212] nkn(x) => Onkn(t)

213 [] x => Onkn(neg p)

252 [213] nkn(x) => Onkn(neg p)

214 [] nkn(x) => Onkn(neg p)

255 [214] nkn(x) => Onkn(neg p)

215 [] nkn(x) => Onkn(t)

256 [215] nkn(x) => Onkn(t)
```

```
  216 [] t => Onkn(neg p)
  257 [216] nkn(t) => Onkn(neg p)
  217 [] t => Onkn(t)
  260 [217] nkn(t) => Onkn(t)
]

Output from snl   ==> New modal rules are
[
]
```

The prover separates and shows all the clauses after the generation of SNL literals:

```
run_tempres_or_otter ==> (1)
Literal - Old Literal clauses
[
  104 [49, 14] (nkn(t) | ~nkn(y))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  46 [12, 2] (nkn(x) | ~tmpp1)
  41 [19, 4] (nkn(neg p) | ~x)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
```

```
  29 [12, mres5] (nkn(x) | ~x)
  28 [19, mres5] (nkn(neg p) | p)
  27 [17, mres5] (nkn(t) | ~t)
]
NewLiteral - All Literal clauses coming from otter
[
  104 [49, 14] (nkn(t) | ~nkn(y))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  46 [12, 2] (nkn(x) | ~tmpp1)
  41 [19, 4] (nkn(neg p) | ~x)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  29 [12, mres5] (nkn(x) | ~x)
  28 [19, mres5] (nkn(neg p) | p)
  27 [17, mres5] (nkn(t) | ~t)
]
SmallNewLiteral - the really new clauses generated by otter
[
]
```

As there were no new literal clauses coming from Otter, step resolution is finished and the prover starts temporal resolution:

```
run_tempres_or_modal ==> (1)
Attempting Breadth-First Temporal Resolution


full_tempres (2) ==> Previous loop paths.
Attempting temporal resolution with p.
[
  8 [] t => O~p
  210 [] x => O~p
]
[
  104 [49, 14] (nkn(t) | ~nkn(y))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  46 [12, 2] (nkn(x) | ~tmpp1)
  41 [19, 4] (nkn(neg p) | ~x)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  29 [12, mres5] (nkn(x) | ~x)
```

```
28 [19, mres5] (nkn(neg p) | p)
27 [17, mres5] (nkn(t) | ~t)
5 [] x => Oy
218 [5] nkn(x) => Onkn(y)
10 [5] nkn(x) => Onkn(y)
223 [10] nkn(x) => Onkn(y)
224 [8] nkn(t) => Onkn(neg p)
15 [8] nkn(t) => Onkn(neg p)
229 [15] nkn(t) => Onkn(neg p)
9 [] t => Ot
230 [9] nkn(t) => Onkn(t)
20 [9] nkn(t) => Onkn(t)
235 [20] nkn(t) => Onkn(t)
209 [] x => Onkn(y)
236 [209] nkn(x) => Onkn(y)
239 [210] nkn(x) => Onkn(neg p)
211 [] x => Ot
244 [211] nkn(x) => Onkn(t)
212 [] x => Onkn(t)
249 [212] nkn(x) => Onkn(t)
213 [] x => Onkn(neg p)
252 [213] nkn(x) => Onkn(neg p)
214 [] nkn(x) => Onkn(neg p)
255 [214] nkn(x) => Onkn(neg p)
215 [] nkn(x) => Onkn(t)
256 [215] nkn(x) => Onkn(t)
216 [] t => Onkn(neg p)
```

```
  257 [216] nkn(t) => Onkn(neg p)
  217 [] t => Onkn(t)
  260 [217] nkn(t) => Onkn(t)
]
```

where the first set is the set of clauses related to the loop (here $\neg p$) and the second set comprises the remaining literal and step clauses. Next, the prover shows the loop that was found:

```
The New Node is [[t], [x]].
The eventuality (for possibly several rules) being considered is p
New Loop detected is [[t], [x]]
Loops previously detected are
    [
     [p, []]
    ]
```

In this example, the loop is given by $t \wedge x$. Then it shows the resolvents obtained from temporal resolution:

```
[
  tmpp2 => O(tmpp2 | p)
  (~x | (tmpp2 | p))
  (~x | (p | ~t))
  tmpp2 => O(~t | p)
  (~x | (p | ~x))
  tmpp2 => O(~x | p)
]
```

The resolvents are numbered and separated into literal and step clauses:

```
otres_allres (2) ==> New literal rules are
[
  264 [] (~x | (tmpp2 | p))
  265 [] (~x | (p | ~t))
  267 [] (~x | p)
]
otres_allres (2) ==> New temporal rules are
[
  263 [] tmpp2 => O(tmpp2 | p)
  266 [] tmpp2 => O(~t | p)
  268 [] tmpp2 => O(~x | p)
]
```

Then, a new literal/modal cycle starts, where the `Old` sets are the clauses obtained in the previous cycle and the `New` sets are the clauses obtained from the temporal resolution.

```
Cycle Number1.
Old Literal Clauses
[
  104 [49, 14] (nkn(t) | ~nkn(y))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  46 [12, 2] (nkn(x) | ~tmpp1)
  41 [19, 4] (nkn(neg p) | ~x)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
```

```
  7 [] (~y | t)
  6 [] (~y | ~p)
  4 [] (~x | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  29 [12, mres5] (nkn(x) | ~x)
  28 [19, mres5] (nkn(neg p) | p)
  27 [17, mres5] (nkn(t) | ~t)
]

Old Modal Clauses
[
  127 [104, 14] ~nkn(t) => k~y
  124 [95, 14] ~nkn(neg p) => k~y
  120 [78, 12] ~nkn(neg p) => k~x
  111 [49, 30] y => ~k~t
  102 [48, 30] y => ~kp
  86 [44, 46] tmpp1 => ~kp
  85 [44, 29] x => ~kp
  74 [43, 2] tmpp1 => ~k~x
  49 [13, 7] nkn(y) => ~k~t
  48 [13, 6] nkn(y) => ~kp
  47 [13, 30] y => ~k~y
  44 [11, 4] nkn(x) => ~kp
  43 [11, 29] x => ~k~x
  39 [18, 28] ~p => ~kp
  36 [16, 27] t => ~k~t
```

```
  14 [5] ~nkn(y) => k~y
  13 [5] nkn(y) => ~k~y
  12 [5] ~nkn(x) => k~x
  11 [5] nkn(x) => ~k~x
  19 [8] ~nkn(neg p) => kp
  18 [8] nkn(neg p) => ~kp
  17 [8] ~nkn(t) => k~t
  16 [8] nkn(t) => ~k~t
]

New Literal Clauses
[
  267 [] (~x | p)
]

New Modal Clauses
[
]
```

Note that the first and second resolvents obatined by temporal resolution are subsumed by the third resolvent. This is why they have been ommited in the set of new clauses. Modal/literal resolution is performed, in this case, until a contradiction with the initial clauses is found:

```
Cycle Number2.
Old Literal Clauses
[
  104 [49, 14] (nkn(t) | ~nkn(y))
  95 [48, 14] (nkn(neg p) | ~nkn(y))
```

```
  78 [44, 12] (nkn(neg p) | ~nkn(x))
  59 [41, 2] (nkn(neg p) | ~tmpp1)
  46 [12, 2] (nkn(x) | ~tmpp1)
  42 [19, 6] (nkn(neg p) | ~y)
  38 [17, 7] (nkn(t) | ~y)
  33 [4, 2] (~p | ~tmpp1)
  7 [] (~y | t)
  6 [] (~y | ~p)
  2 [] (~tmpp1 | x)
  30 [14, mres5] (nkn(y) | ~y)
  28 [19, mres5] (nkn(neg p) | p)
  27 [17, mres5] (nkn(t) | ~t)
]

Old Modal Clauses
[
  127 [104, 14] ~nkn(t) => k~y
  124 [95, 14] ~nkn(neg p) => k~y
  120 [78, 12] ~nkn(neg p) => k~x
  111 [49, 30] y => ~k~t
  102 [48, 30] y => ~kp
  86 [44, 46] tmpp1 => ~kp
  74 [43, 2] tmpp1 => ~k~x
  49 [13, 7] nkn(y) => ~k~t
  48 [13, 6] nkn(y) => ~kp
  47 [13, 30] y => ~k~y
  44 [11, 4] nkn(x) => ~kp
```

```
  39 [18, 28] ~p => ~kp
  36 [16, 27] t => ~k~t
  14 [5] ~nkn(y) => k~y
  13 [5] nkn(y) => ~k~y
  12 [5] ~nkn(x) => k~x
  11 [5] nkn(x) => ~k~x
  19 [8] ~nkn(neg p) => kp
  18 [8] nkn(neg p) => ~kp
  17 [8] ~nkn(t) => k~t
  16 [8] nkn(t) => ~k~t
]

New Literal Clauses
[
  280 [267, 4] ~x
  279 [267, 2] (p | ~tmpp1)
]

New Modal Clauses
[
  269 [267, 18] nkn(neg p) => ~kx
  270 [267, 11] nkn(x) => ~k~p
  271 [267, 39] ~p => ~kx
  273 [267, 44] nkn(x) => ~kx
  274 [267, 48] nkn(y) => ~kx
  275 [267, 74] tmpp1 => ~k~p
  277 [267, 86] tmpp1 => ~kx
```

```
  278 [267, 102] y => ~kx
]
```

This is shown in the last set of clauses:

```
New literal clauses include (neg start).
[
  283 [280, 11] ~nkn(x)
  293 [279, 33] ~tmpp1
  294 [269, 19] (nkn(neg x) | ~nkn(neg p))
  339 [274, 14] (nkn(neg x) | ~nkn(y))
]
```

Finally, the prover provides the messages saying that the formula is unsatisifiable and gives the statistics:

```
Have generated false in initial clauses.
Formula is unsatisfiable (1).


Statistics


CPUTime:0.18
Inferences:274347
Local Stack:92496
Global Stack:41032
Trail Stack:6652


Number of clauses generated: 351
```

Appendix B

Examples Code

```
ex354:-otres([start imp x,
              x imp sometime p,
              neg x or neg p,
              x imp next y,
              neg y or neg p,
              neg y or t,
              t imp next neg p,
              t imp next t]).

ex5211:-otres([start imp x,
               x imp k y,
               y imp sometime p,
               x imp k neg p,
               x imp neg k neg z,
               z imp next w,
               neg w or neg p,
               neg w or t,
               t imp next neg p,
```

```
               t imp next t]).

ex5212:-otres([start imp x,
               x imp sometime y,
               y imp k p,
               x imp k neg p,
               x imp next z,
               neg z or w,
               neg z or t,
               t imp next w,
               t imp next t,
               w imp neg k p]).

ex621:-otres([start imp x,
              x imp k y,
              y imp next p,
              x imp next z,
              z imp neg k p]).

ex7121:-otres([start imp x,
               x imp next y,
               y imp k p,
               x imp neg k neg z,
               z imp next neg p]).

ex7122:-otres([start imp x,
               x imp next y,
```

```
               y imp k p,
               x imp neg k neg v,
               neg v or t,
               neg v or u,
               t imp next (w or neg p),
               u imp next (neg w or neg p)]).

ex7123:-otres([start imp x,
               neg x or z or y,
               neg x or z or w,
               w imp next (z or y),
               w imp next (z or w),
               x imp sometime z,
               z imp k p2,
               y imp k p1,
               x imp neg k neg r,
               neg r or t or s,
               neg r or t or u,
               u imp next (t or s),
               u imp next (t or u),
               t imp neg k p1,
               t imp neg k p2,
               s imp neg k p2]).

ex81:-otres([neg(k always p imp always k p)]).
ex82:-otres([neg(always k p imp k always p)]).
ex83:-otres([(k sometime p and k neg p) imp k next sometime p]).
```