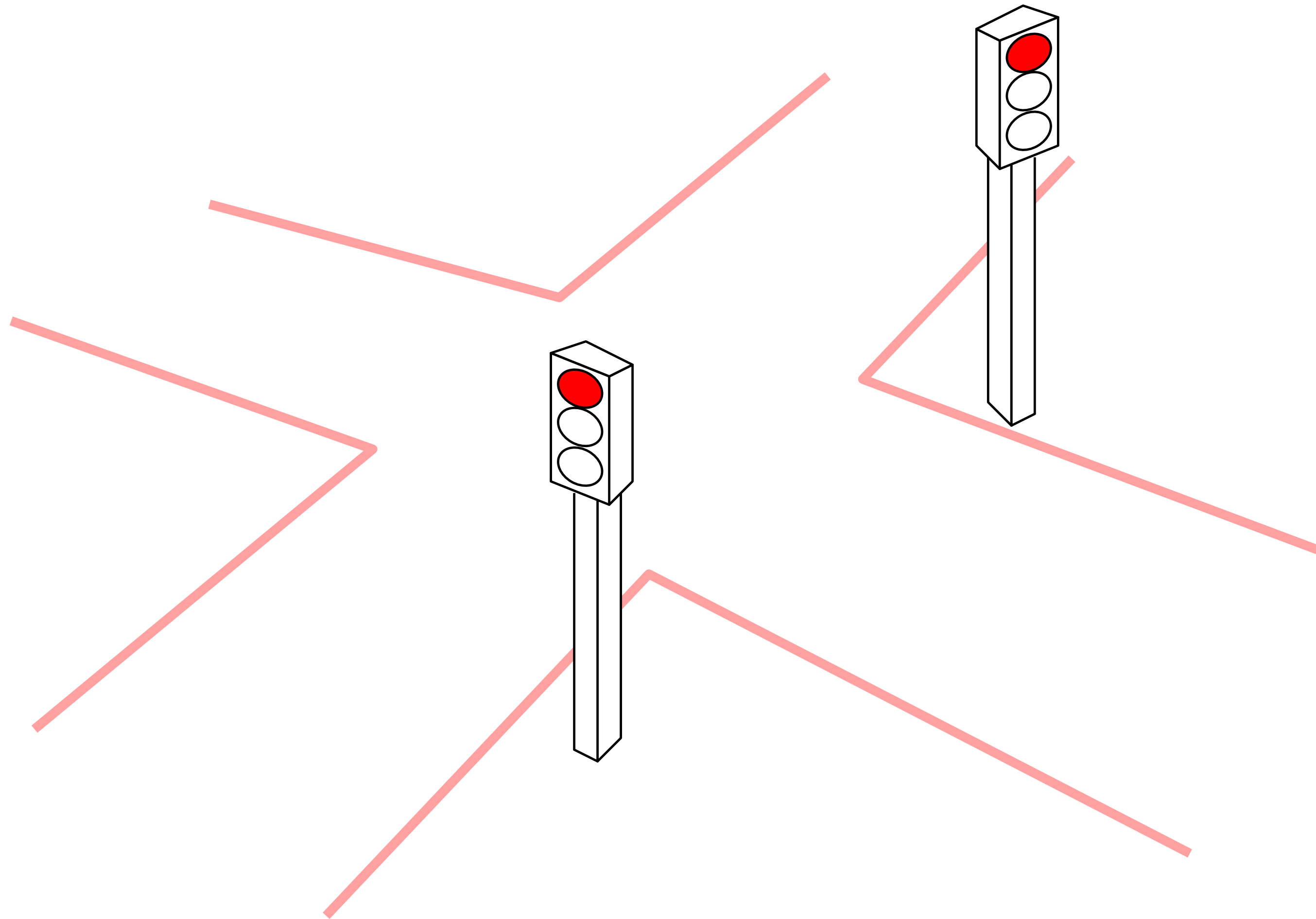
Model Checking

- check **algorithmically** temporal / sequential properties
  - systems are originally **finite state**
  - simple model: finite state automaton
- **comparison** of automata can be seen as model checking
  - check that the output streams of two finite state systems “match”
  - process algebra: simulation and bisimulation checking
- **temporal logics** as specification mechanism
  - safety, liveness and more general temporal operators, fairness

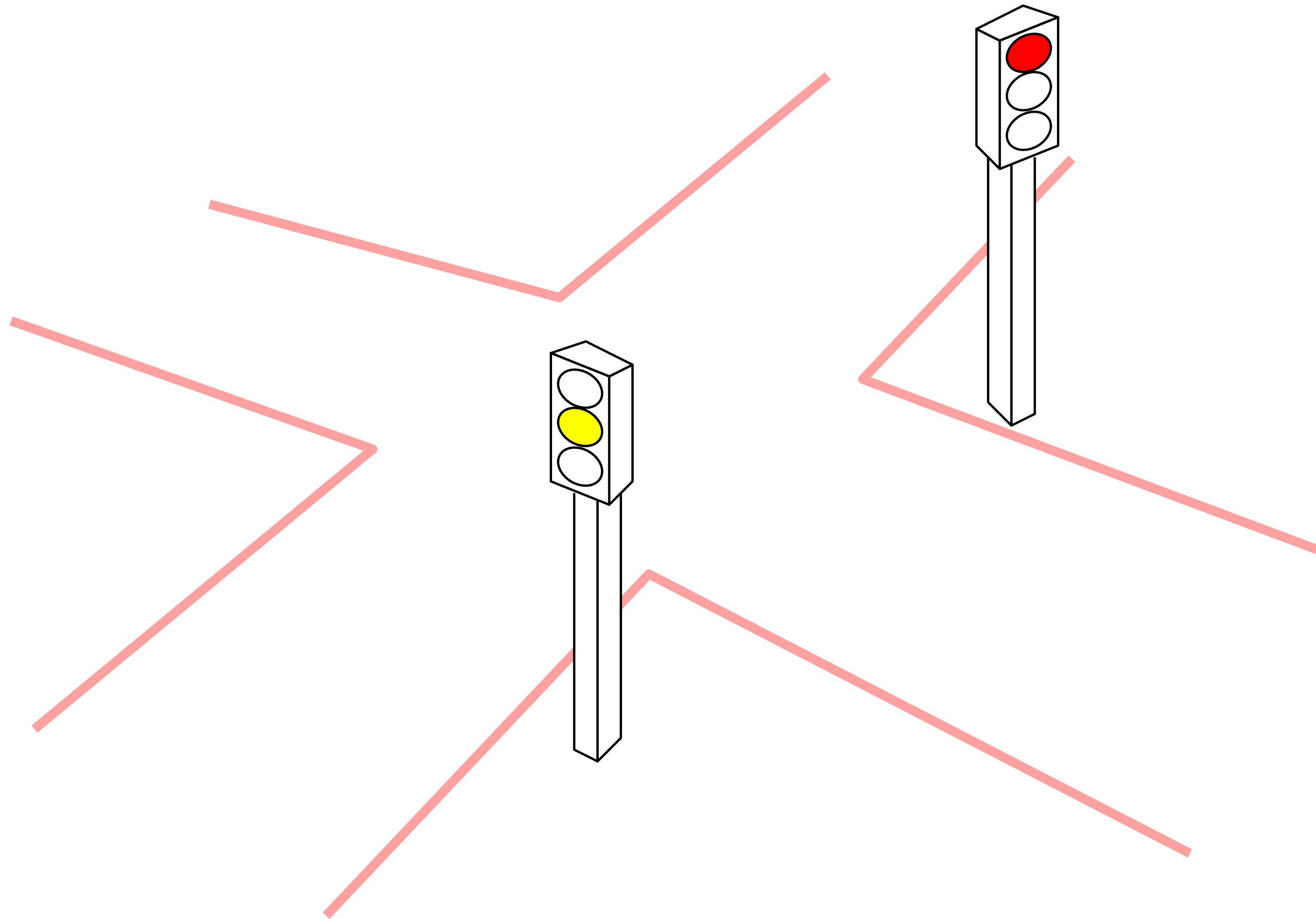
Model Checking for Infinite Systems

- fixpoint algorithms with symbolic representations:
  - timed automata (clocks)
  - hybrid automata (differential equations)
  - termination guaranteed if finite quotient structure exists
- simply run model checker *for some time*, e.g. Java Pathfinder
- run time verification
  1. example: add checker synthesized from temporal spec
  2. example: run all schedules for one test case
- check programs (incl. loops and recursion) over finite domains, e.g. SLAM

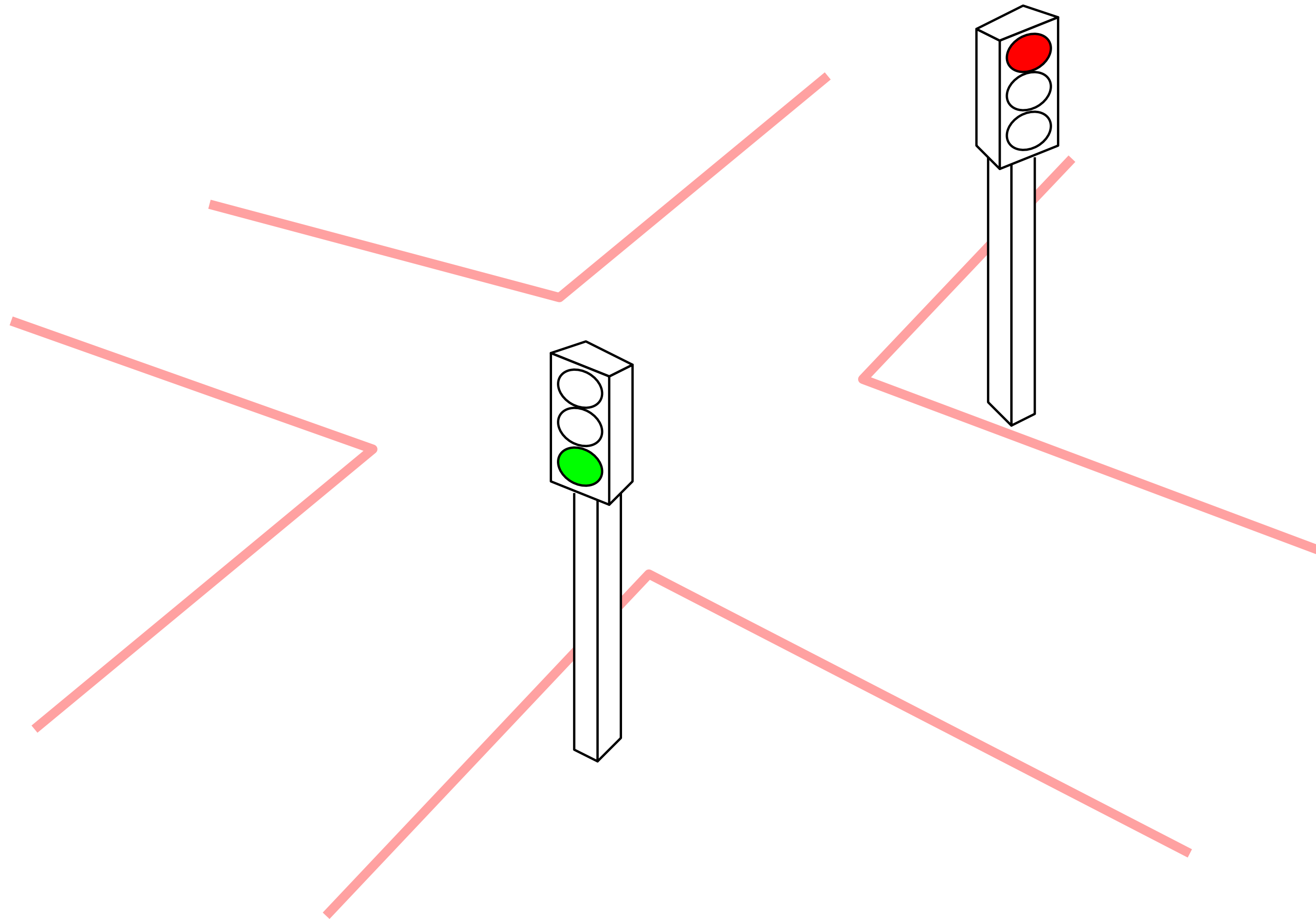
Traffic Light Controller (TLC)

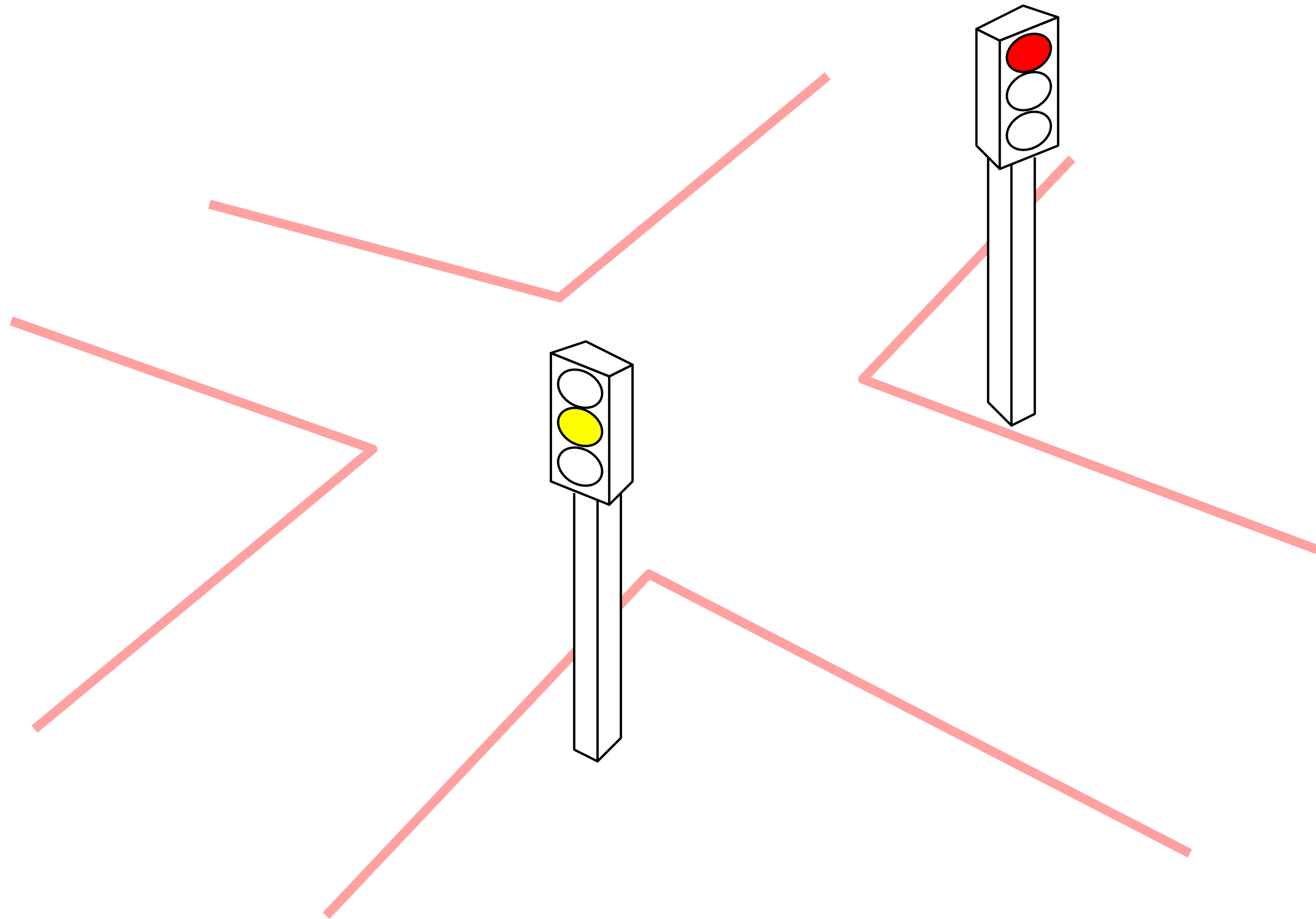


Traffic Light Controller (TLC)

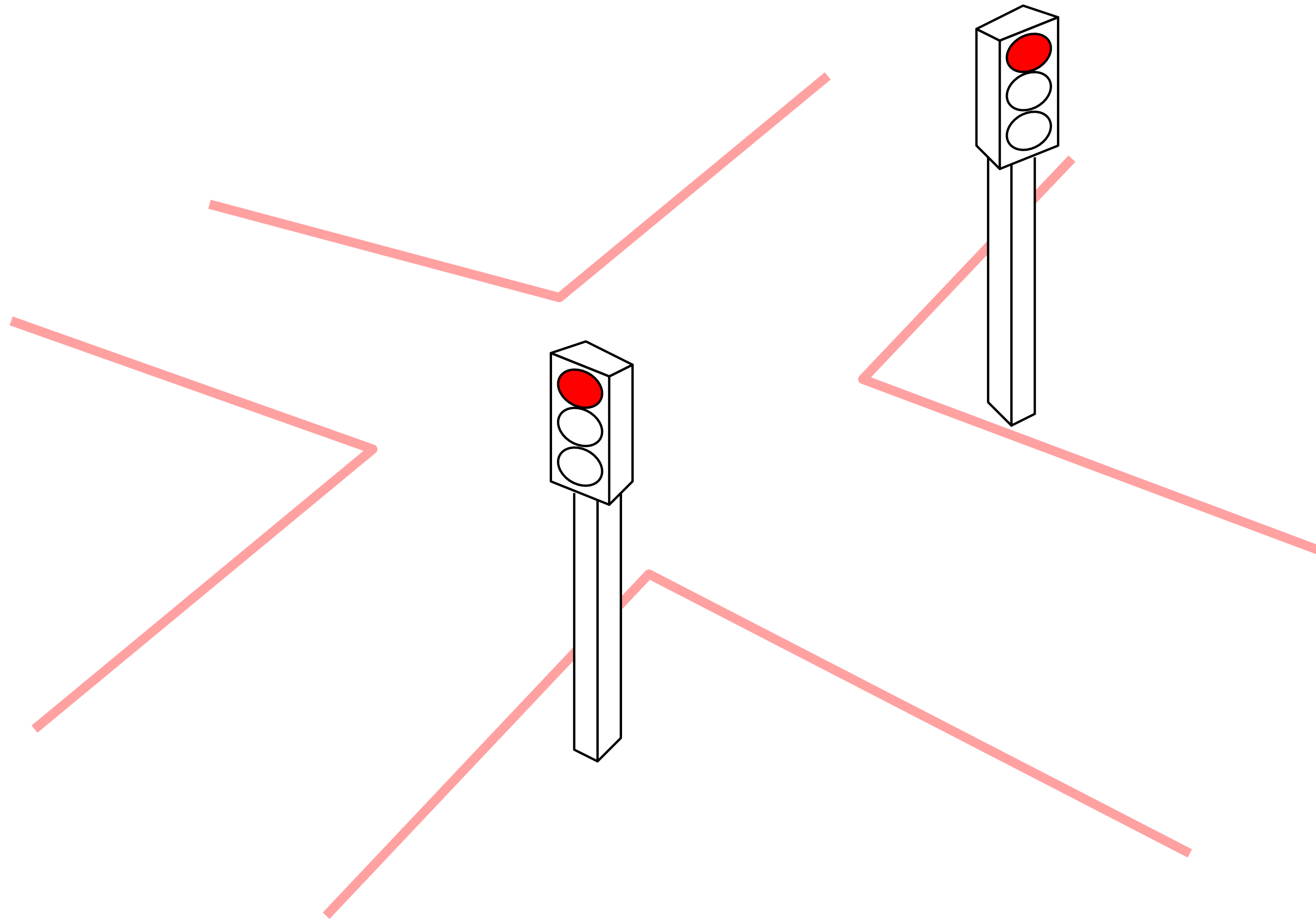


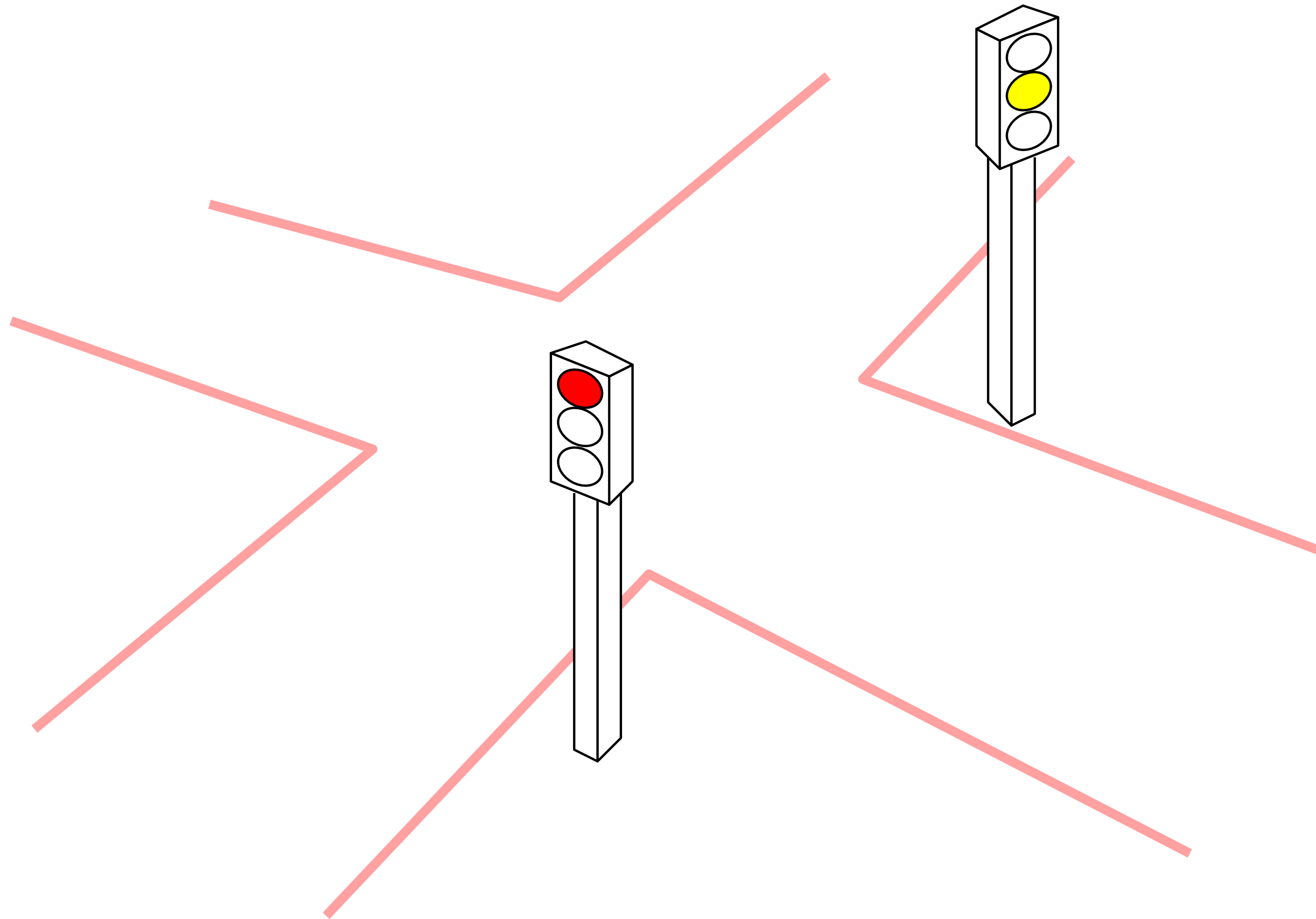
Traffic Light Controller (TLC)



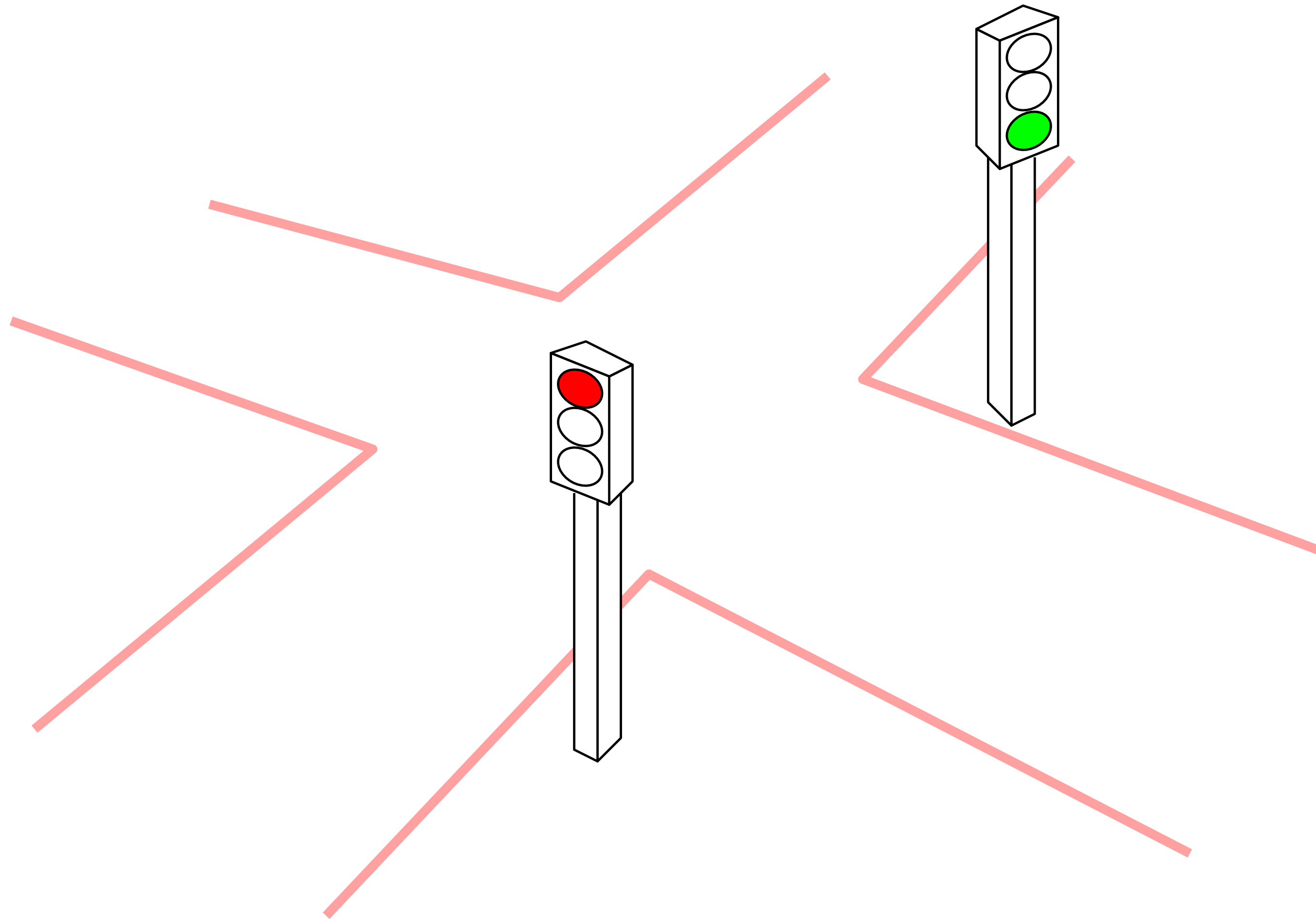


Traffic Light Controller (TLC)

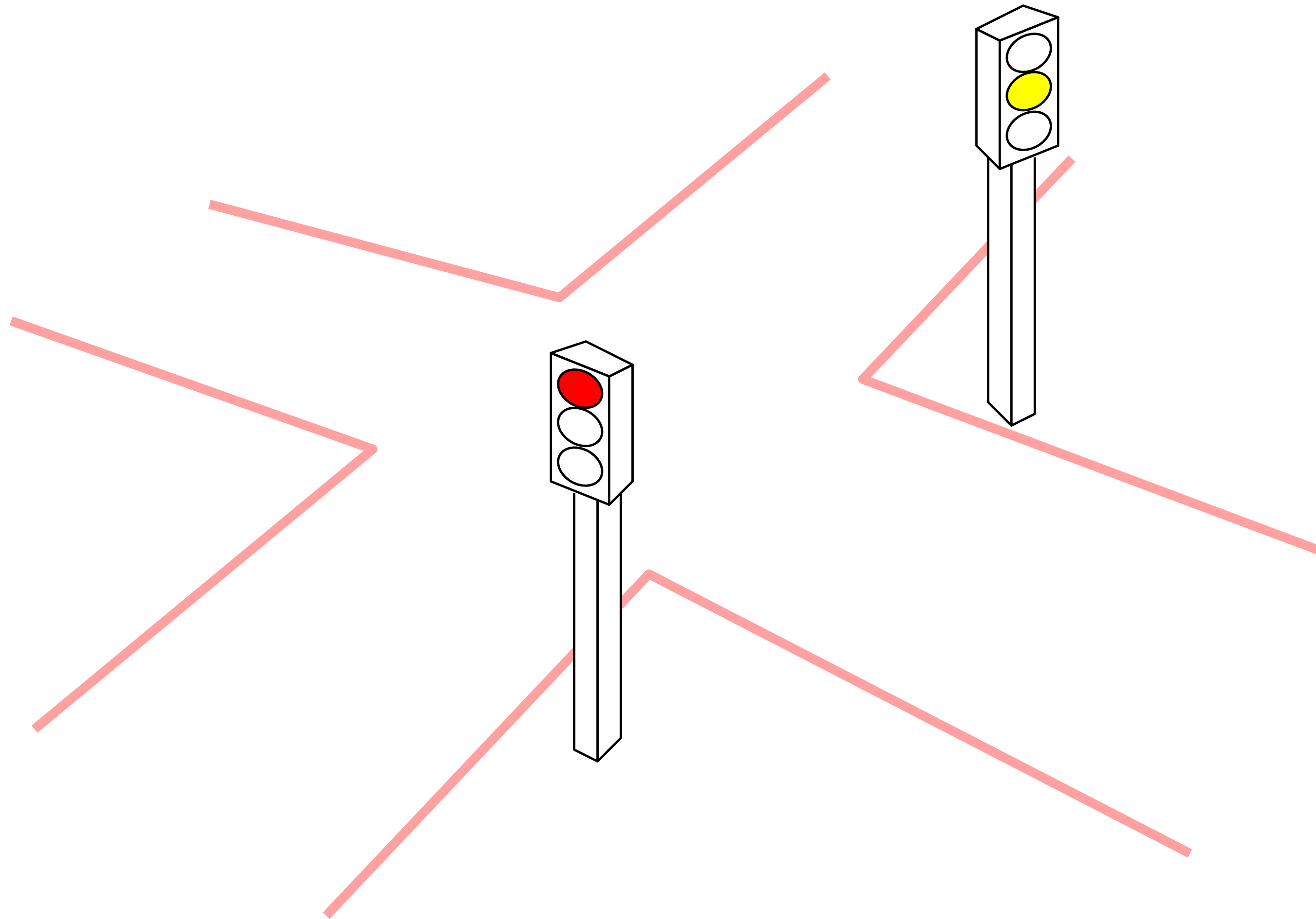




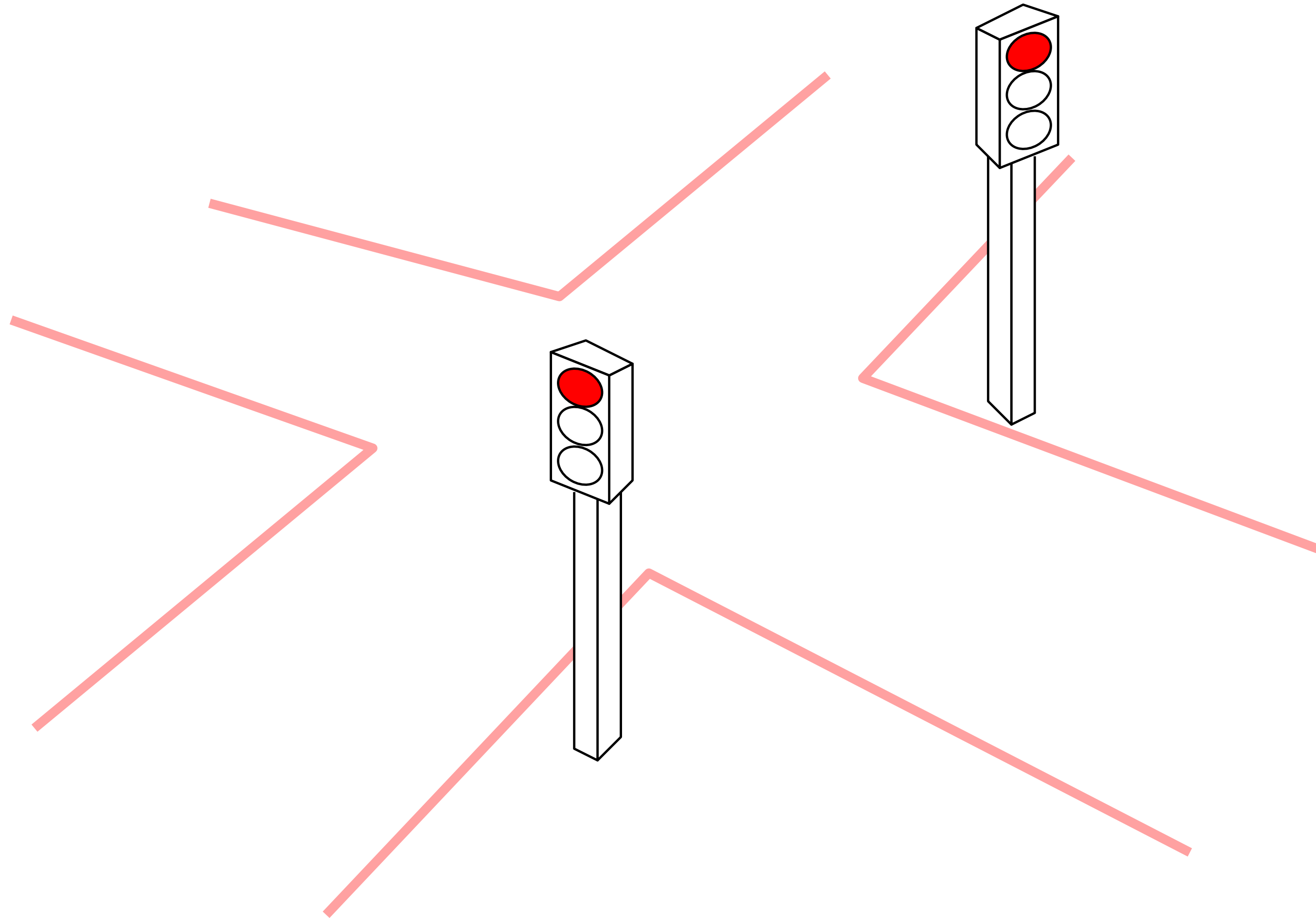
Traffic Light Controller (TLC)

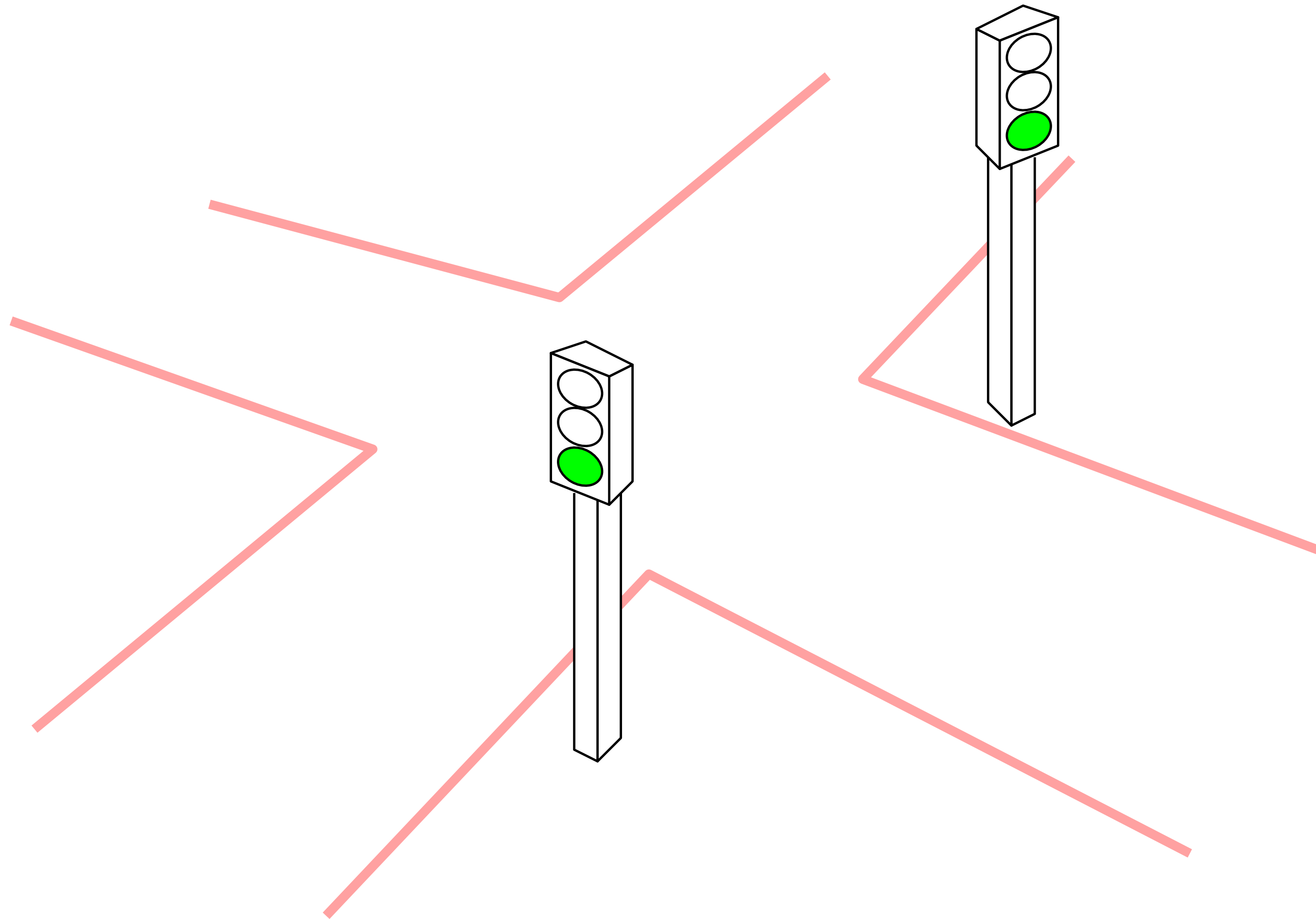


Traffic Light Controller (TLC)



Traffic Light Controller (TLC)





the two traffic lights should never show a green light at the same time

State Space

- state space is the set of assignments to variables of the system
  - state space is finite if the **range** of variables is finite
  - this notion works for inifinite state spaces as well
- TLC example:
  - single assignment $\sigma\colon \{southnorth, eastwest\} \to \{green, yellow, red\}$
  - set of assignments is isomorphic to $\{green, yellow, red\}^2$
  - eg state space is isomorphic to the crossproduct of variable ranges
- not all states are reachable: $(green, green)$

Safety

- safety properties specify **invariants** of the system

- simple generic algorithm for checking safety properties:

  1. iteratively generate all reachable states

  2. check for violation of invariant for newly reached states

  3. terminate if all newly reached states can be found

- compare with **assertions**

  – used in run time checking: `assert` in C and VHDL

  – contract checking: `require`, `ensure`, etc. in Eiffel

```
MODULE trafficlight (enable)
VAR
  light : { green, yellow, red };
  back : boolean;
ASSIGN
  init (light) := red;
  next (light) :=
    case
      light = red & !enable : red;
      light = red & enable : yellow;
      light = yellow & back : red;
      light = yellow & !back : green;
      1 : yellow;
    esac;
  next (back) :=
    case
      light = red & enable : 0;
      light = green : 1;
      1 : back;
    esac;
MODULE main
VAR
  southnorth : trafficlight (1);
  eastwest : trafficlight (1);
SPEC
  AG !(southnorth.light = green & eastwest.light = green)
```

- symbolic model checker implemented by Ken McMillan at CMU (early 90'ies)

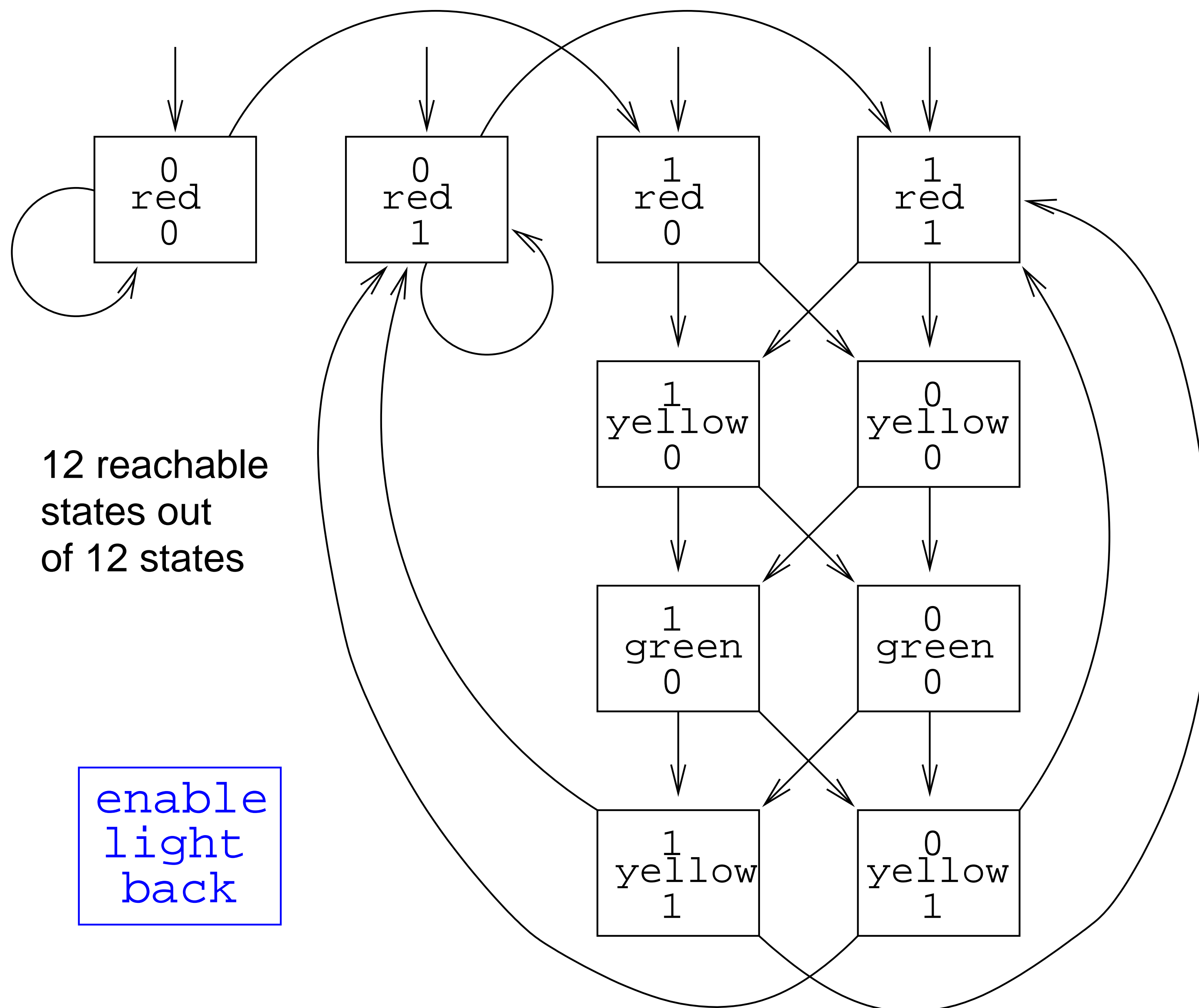
- input language: finite models + temporal specification

- hierarchical description, similar to hardware description language (HDL)

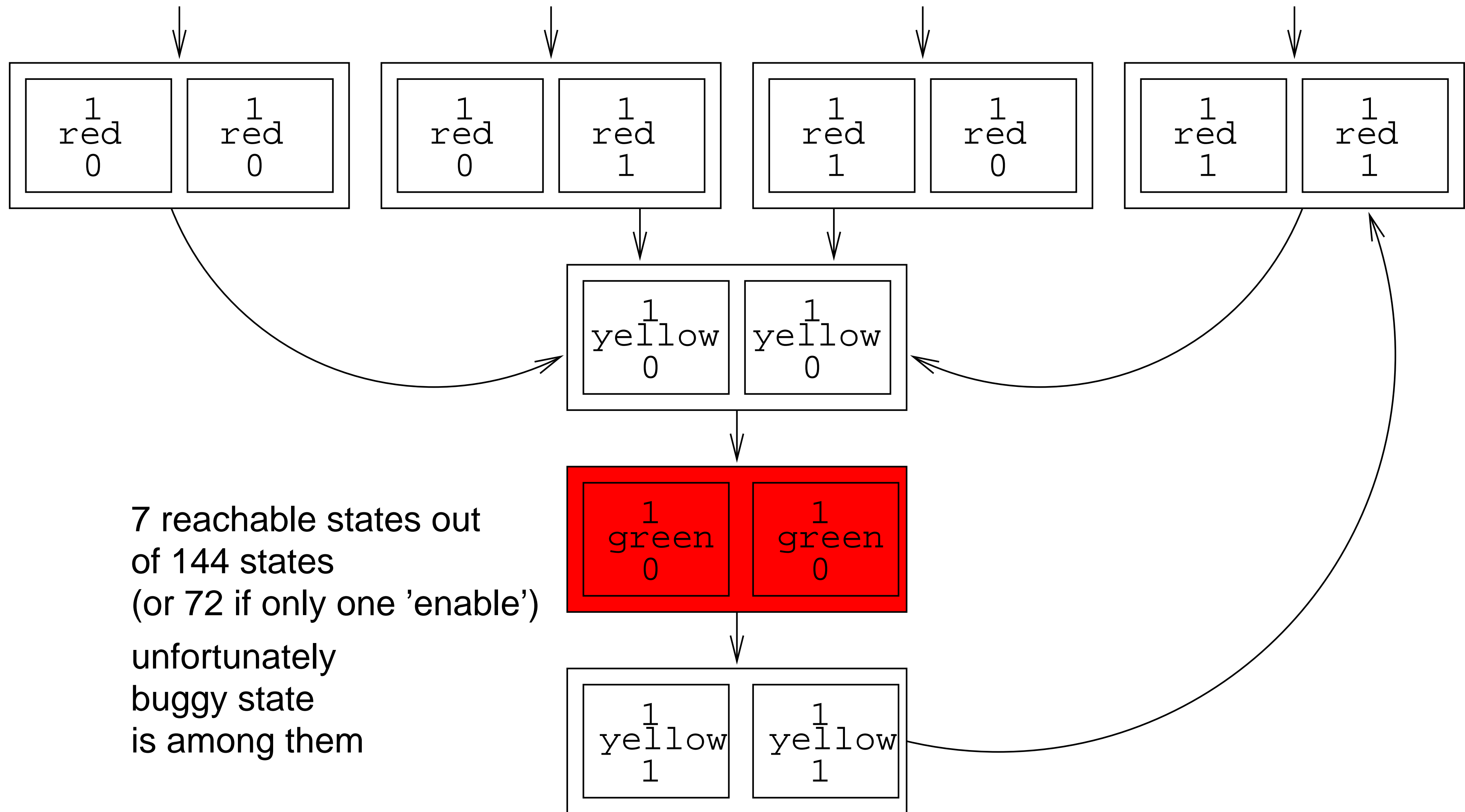
- integer and enumeration types, arithmetic operations

- heavily relies on the data structure Binary Decision Diagrams (BDDs)

Reachable States of One Traffic Light



Reachable States of Unsafe TLC

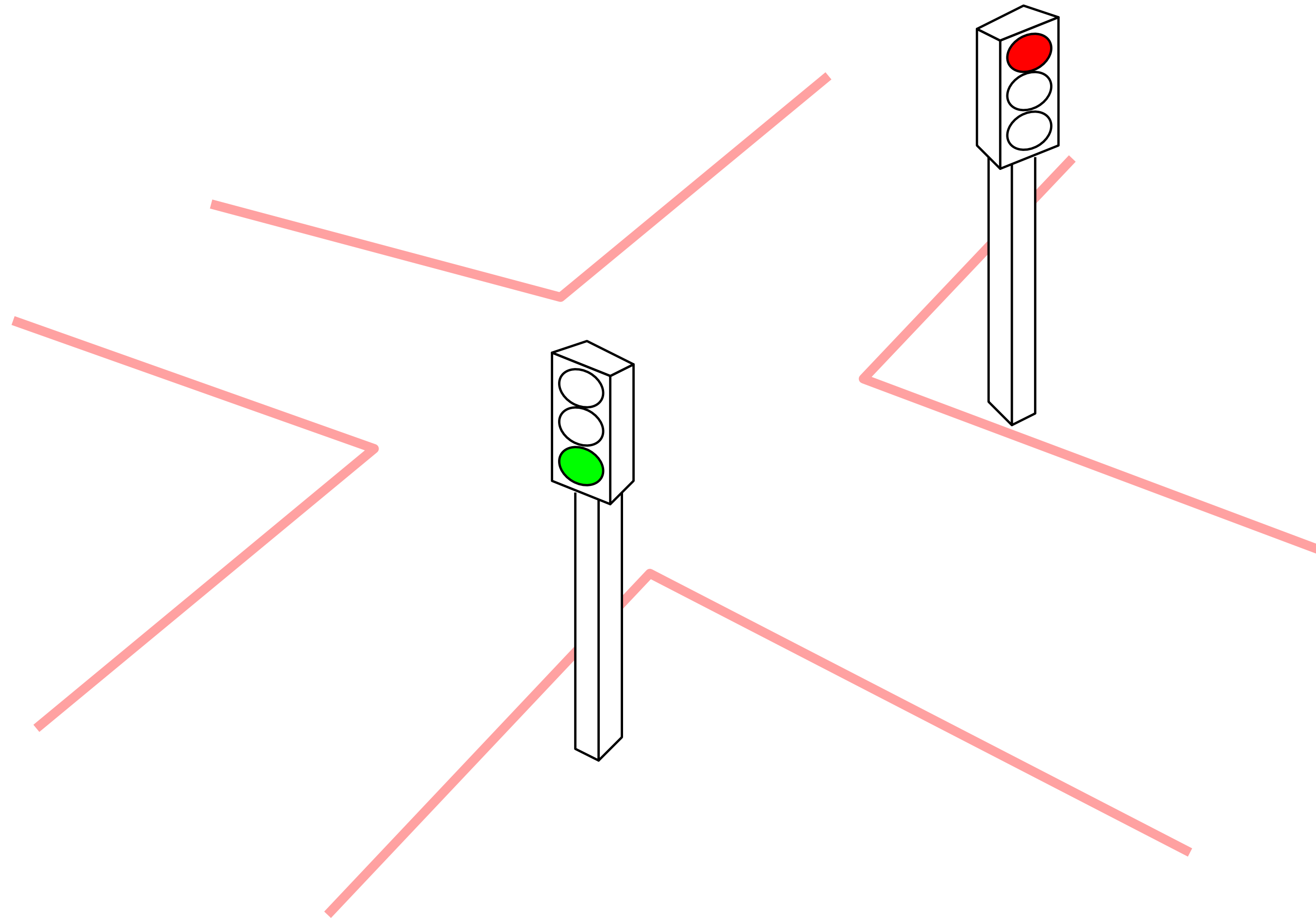


7 reachable states out
of 144 states
(or 72 if only one ’enable’)

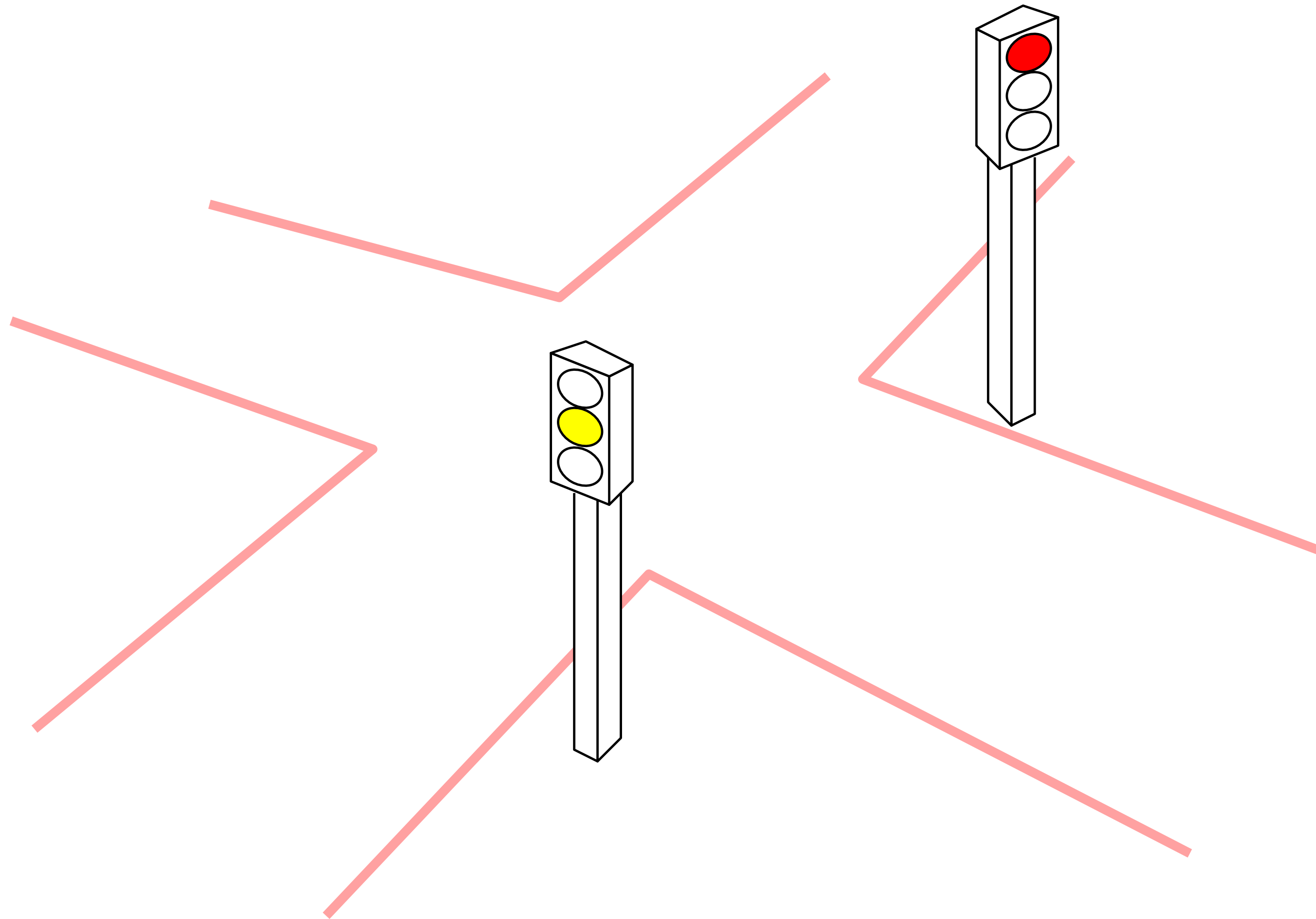
unfortunately
buggy state
is among them

```
MODULE main
VAR
  turn : { ew, sn };
  southnorth : trafficlight (enablesouthnorth);
  eastwest : trafficlight (enableeastwest);
DEFINE
  enableeastwest := southnorth.light = red & turn = ew;
  enablesouthnorth := eastwest.light = red & turn = sn;
SPEC
  AG !(southnorth.light = green & eastwest.light = green)
```

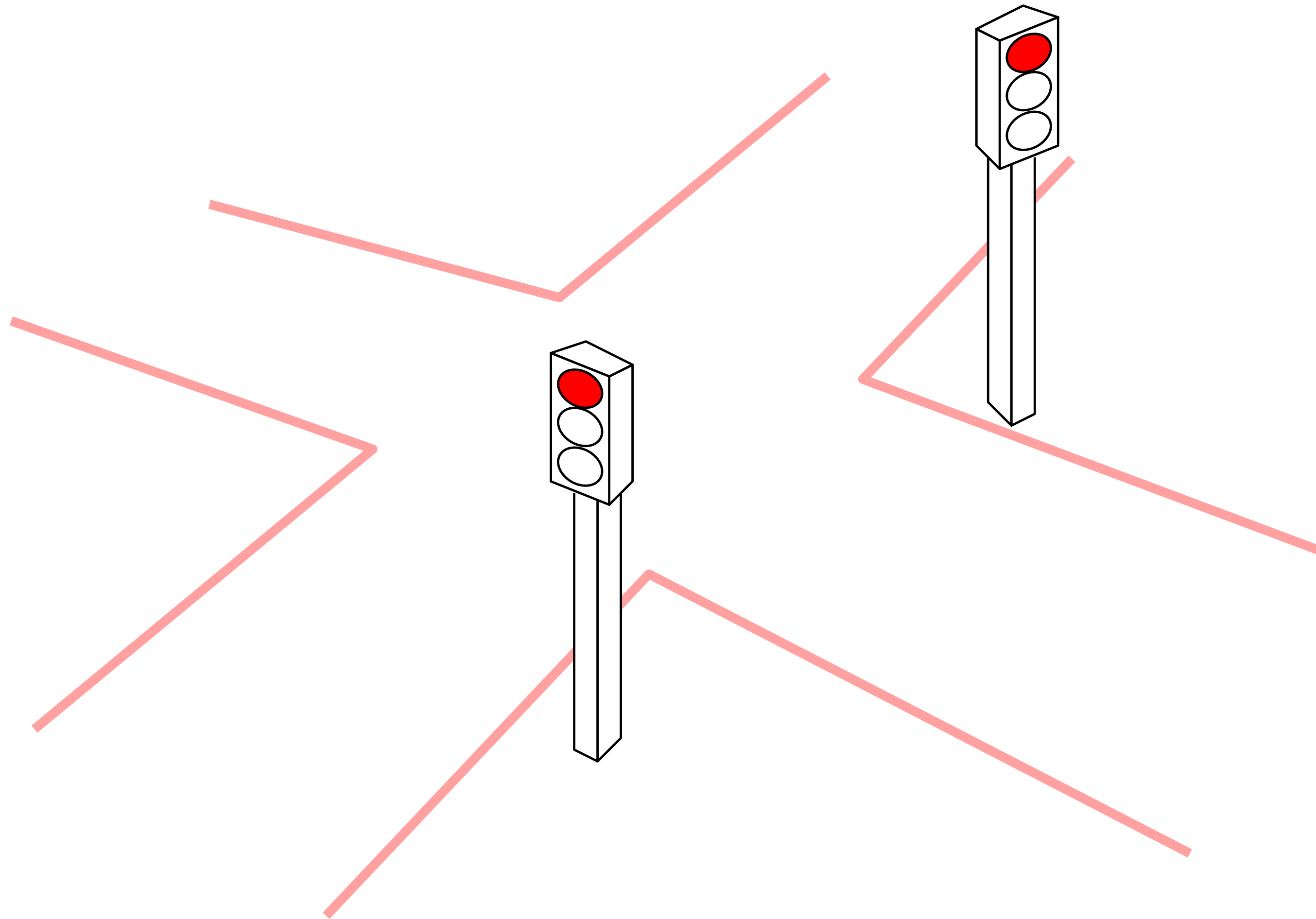
idea: disable traffic light as long the other is not red and its not the others turn



traffic lights showing red should eventually show green



traffic lights showing red should eventually show green



traffic lights showing red should eventually show green

Boolean Encoding

- compilation of finite model into pure propositional domain
- first step is to flatten the hierarchy
  - recursive instantiation of all submodules
  - name and parameter substitution
  - may increase program size exponentially
- second step is to encode variables with boolean variables

light		light@1	light@0
green	$\mapsto$	0	0
yellow	$\mapsto$	0	1
red	$\mapsto$	1	0

Result of Boolean Encoding

- initial state predicate I represented as boolean formula

  `!eastwest.light@0 & eastwest.light@1`

  (equivalent to `init(eastwest.light) := red`)

- transition relation T represented as boolean formula

- encoding of atomic predicates p as boolean formulae

  `!eastwest.light@1 & !eastwest.light@0`

  (equivalent to `eastwest.light != green`)