Kind 2 Qualification Kit

Adrien Champion Alain Mebsout Christoph Sticksel
Cesare Tinelli
The University of Iowa

August 7, 2015

Contents

1 User Manual[1]

Kind 2 is a multi-engine, parallel, SMT-based automatic model checker for safety properties of Lustre programs.

Kind 2 takes as input a Lustre file annotated with properties to be proven invariant (see Lustre syntax), and outputs which of the properties are true for all inputs, as well as an input sequence for those properties that are falsified. To ease processing by front- end tools, Kind 2 can output its results in XML format.

[1]The most current version of this user manual can be obtained with Kind 2's sources in the subdirectory `doc/usr` or online at `https://github.com/kind2-mc/kind2/tree/master/doc/usr/content`.

By default Kind 2 runs a process for bounded model checking (BMC), a process for k-induction, two processes for invariant generation, and a process for IC3 in parallel on all properties simultaneously. It incrementally outputs counterexamples to properties as well as properties proved invariant.

The following command-line options control its operation (run `kind2 --help` for a full list). See also the description of the techniques for configuration examples and more details on each technique.

`--enable {BMC|IND|INVGEN|INVGENOS|IC3}` Select model checking engines

By default, all three model checking engines are run in parallel. Give any combination of `--enable BMC`, `--enable IND` and `--enable IC3` to select which engines to run. The option `--enable BMC` alone will not be able to prove properties valid, choosing `--enable IND` only will not produce any results. Any other combination is sound (properties claimed to be invariant are indeed invariant) and counterexample-complete (a counterexample will be produced for each property that is not invariant, given enough time and resources).

`--timeout_wall <int>` (default `0` = none) – Run for the given number of seconds of wall clock time

`--timeout_virtual <int>` (default `0` = none) – Run for the given number of seconds of CPU time

`--smtsolver {CVC4|Yices|Z3}` (default `Z3`) – Select SMT solver

The default is `Z3`, but see options of the `./build.sh` script to override at compile time

`--cvc4_bin <file>` – Executable for CVC4

`--yices_bin <file>` – Executable for Yices

`--z3_bin <file>` – Executable for Z3

`-v` Output informational messages

`-xml` Output in XML format

1.1 Obtaining Kind 2

Kind 2 is distributed under the Apache 2 license. It can be obtained at the following url:

`http://kind2-mc.github.io/kind2/`

It is available as a source distribution (formats `tar.gz` or `zip`) or as pre-compiled 64-bit binaries for linux and Mac OS X.

1.2 Requirements

- Linux or Mac OS X,
- OCaml 4.02 or later,
- Camlp4

- Menhir parser generator, and
- a supported SMT solver
  - CVC4,
  - Yices 2, or
  - Yices 1
  - Z3 (presently recommended),

1.3 Building and installing

You need to run first

```
./autogen.sh
```

By default, `kind2` will be installed into `/usr/local/bin`, an operation for which you usually need to be root. Call

```
./build.sh --prefix=<path>
```

to install the Kind 2 binary into `<path>/bin`. You can omit the option to accept the default path of `/usr/local/bin`.

The ZeroMQ and CZMQ libraries, and OCaml bindings to CZMQ are distributed with Kind 2. The build script will compile and link to those, ignoring any versions that are installed on your system.

If it has been successful, call

```
make install
```

to install the Kind 2 binary into the chosen location. If you need to pass options to the configure scripts of any of ZeroMQ, CZMQ, the OCaml bindings or Kind 2, add these to the `build.sh` call. Use `./configure --help` after `autogen.sh` to see all available options.

You need a supported SMT solver on your path when running `kind2`.

You can run tests to see if Kind 2 has been built correctly. To do so run

```
make test
```

You can pass arguments to Kind 2 with the `ARGS="..."` syntax. For instance

```
make ARGS="--enable PDR" test
```

1.4 Documentation

You can generate the user documentation by running `make doc`. This will generate a `pdf` document in `doc/` corresponding to the markdown documentation available on the GitHub page.

To generate the documentation, you need

- a GNU version of `sed` (`gsed` on OSX), and
- Pandoc.

1.5 Techniques

This section presents the techniques available in Kind 2. How they work and how they can be tweaked through options:

- k-induction
- invariant generation
- IC3

1.5.1 K-induction

K-induction is a well-known technique for the verification of transition systems. A k-induction engine is composed of two parts: *base* and *step*. Base performs bounded model checking on the properties, *i.e.* checks the **base case**. Step checks whether it is possible to reach a violation of one of the properties from a trace of states satisfying them: the **inductive step**.

In Kind 2 base and step run in parallel, and can be enabled separately. Running step alone with

```
kind2 --enable IND <file>
```

will not yield anything interesting, as step cannot falsify properties nor prove anything without base. To run the actual k-induction engine, you must enable base (`BMC`) and step (`IND`):

```
kind2 --enable BMC --enable IND <file>
```

Options

K-induction can be tweaked with the following options.

`--bmc_max <int>` (default `0`) – sets an upper bound on the number of unrolling base and step will perform. `0` is for unlimited.

`--ind_compress <bool>` (default `false`) – activates path compression in step, **i.e.** counterexamples with a loop will be dismissed. You can activate several path compression strategies:

- `--ind_compress_equal <bool>` (default `true`) – compresses states if they are equal modulo inputs
- `--ind_compress_same_succ <bool>` (default `false`) – compresses states if they have the same successors (experimental)
- `--ind_compress_same_pred <bool>` (default `false`) – compresses states if they have the same predecessors (experimental)

`--ind_lazy_invariants <bool>` (default `false`) – deactivates eager use of invariants in step. Instead, when a step counterexample is found each invariant is evaluated on the model until one blocks it. The invariant is then asserted to block the counterexample, and step starts a new check-sat.

1.5.2 Invariant Generation

The invariant generation technique currently implemented in Kind 2 is an improved version of the one implemented in PKind. It works by instantiating templates on a set of terms provided by a syntactic analysis of the system.

The main improvement is that in Kind 2, invariant generation is modular. That is to say it can attempt to discover invariants for subnodes of the top node. The idea is that looking at small components and discovering invariants for them provides results faster than analyzing the system monolithically. To disable the modular behavior of invariant generation, use the option `--invgen_top_only true`.

There are two invariant generation techniques: one state (OS) and two state (TS). The former will only look for invariants between the state variables in the current state, while the latter tries to relate the current state with the previous state. The two are separated because as the system grows in size, two state invariant generation can become very expensive.

The one state and two state variants can be activated with `--enable INVGENOS` and `--enable INVGEN` respectively.

Note that, in theory, two state invariant generation is strictly more powerful than the one state version, albeit slower, since two state can also discover one state invariants. When both variants are running, Kind 2 optimizes two state invariant generation by forcing it to look only for two state invariants.

The bottom line is that running *i)* only two state invariant generation or *ii)* one state and two states will discover the same invariants. In the case of *i)* the same techniques seeks both one state and two state invariants at the same time, which

is slower than *ii)* where one state and two state invariants are sought by different processes running in parallel.

Options

Invariant generation can be tweaked using the following options. Note that this will affect both the one state and two state process if both are running.

`--invgen_prune_trivial <bool>` (default `true`) – when invariants are discovered, do not communicate the ones that are direct consequences of the transition relation.

`--invgen_max_succ <int>` (default `1`) – the number of unrolling to perform on subsystems before moving on to the next one in the hierarchy.

`--invgen_lift_candidates <bool>` (default `false`) – if true, then candidate terms generated for subsystems will be lifted to their callers. **Warning** this might choke invariant generation with a huge number of candidates for large enough systems.

`--invgen_mine_trans <bool>` (default `false`) – if true, the transition relation will be mined for candidate terms. Can make the set of candidate terms untractable.

`--invgen_renice <int>` (only positive values) – the bigger the parameter, the lower the priority of invariant generation will be for the operating system.

Lock Step K-induction

Another improvement on the PKind invariant generation is the way the search for a k-induction proof of the candidate invariants is performed. In PKind, a bounded model checking engine is run up to a certain depth `d` and discovers falsifiable candidate invariants. The graph used to produce the potential invariants is refined based on this information. Once the bound on the depth is reached, an inductive step instance looks for actual invariants by unrolling up to `d`.

In Kind 2, base and step are performed in lock step. Once the candidate invariant graph has been updated by base for some depth, step runs at the same depth and broadcasts the invariants it discovers to the whole framework. It is thus possible to generate invariants earlier and thus speed up the whole analysis.

1.5.3 IC3

IC3/PDR is a recent technique by Aaron Bradley. IC3 alone can falsify and prove properties. To enable nothing but IC3, run

```
kind2 --enable IC3 <file>
```

The challenge when lifting IC3 to infinite state systems is the pre-image computation. If the input problem is in linear integer arithmetic, Kind 2 performs a fast approximate quantifier elimination. Otherwise, the quantifier elimination is delegated to an SMT solver, which is at this time only possible with Z3.

Options

`--ic3_qe {cooper|Z3}` (default `cooper`) – select the quantifier elimination strategy: `cooper` (default) for the built-in approximate method, `Z3` to delegate to the SMT solver. If the problem is not in linear integer arithmetic, `cooper` falls back to `Z3`.

`--ic3_check_inductive <bool>` (default `true`) – Check if a blocking clause is inductive and communicate it as an invariant to concurrent verification engines.

`--ic3_block_in_future <bool>` (default `true`) – Block each clause not only in the frame it was discovered, but also in all higher frames.

`--ic3_fwd_prop_non_gen <bool>` (default `true`) – Attempt forward propagation of clauses before inductive generalization.

`--ic3_fwd_prop_ind_gen <bool>` (default `true`) – Inductively generalize clauses after forward propagation.

`--ic3_fwd_prop_subsume <bool>` (default `true`) – Check syntactic subsumption of forward propagated clauses

1.6 Lustre Input

Lustre is a functional, synchronous dataflow language. Kind 2 supports most of the Lustre V4 syntax and some elements of Lustre V6. See the file `./examples/syntax-test.lus` for examples of all supported language constructs.

1.6.1 Syntax extensions

To specify a property to verify in a Lustre node, add the following in the body (*i.e.* between keywords `let` and `tel`) of the node:

```
--%PROPERTY φ ;
```

where ϕ is a boolean Lustre expression.

Kind 2 only analyzes what it calls the *top node*. By default, the top node is the last node in the file. To force a node to be the top node, add

```
--%MAIN ;
```

to the body of that node.

You can also specify the top node in the command line arguments, with

```
kind2 --lustre_main <node_name> ...
```

1.6.2 Example

The following example declares two nodes `greycounter` and `intcounter`, as well as an *observer* node `top` that calls these nodes and verifies that their outputs are the same. The node `top` is annotated with `--%MAIN ;` which makes it the *top node* (redundant here because it is the last node). The line `--%PROPERTY OK;` means we want to verify that the boolean stream `OK` is always true.

```
node greycounter (reset: bool) returns (out: bool);
var a, b: bool;
let
  a = false -> (not reset and not pre b);
  b = false -> (not reset and pre a);
  out = a and b;
tel

node intcounter (reset: bool; const max: int) returns (out: bool);
var t: int;
let
  t = 0 -> if reset or pre t = max then 0 else pre t + 1;
  out = t = 2;
tel

node top (reset: bool) returns (OK: bool);
var b, d: bool;
let
  b = greycounter(reset);
  d = intcounter(reset, 3);
  OK = b = d;
  --%MAIN ;
  --%PROPERTY OK;
tel
```

Kind 2 produces the following on standard output when run with the default options (`kind2 <file_name.lus>`):

```
kind2 v0.8.0

<Success> Property OK is valid by inductive step after 0.182s.

status of trans sys
------------------------------------------------------------------------------
Summary_of_properties:
```

```
OK: valid
```

We can see here that the property `OK` has been proven valid for the system (by k-induction).

1.7 Using Kind 2

2 Verifying Safety Properties of Lustre Programs with Kind 2: a Tutorial

2.1 Traffic Light System

2.1.1 Description

In this section we model and verify an implementation of a traffic light controller in Lustre. We consider a bi-directional traffic lane with a light at a pedestrian crossing like in Figure 1. The pedestrian can request access to crossing the street by pressing a button.

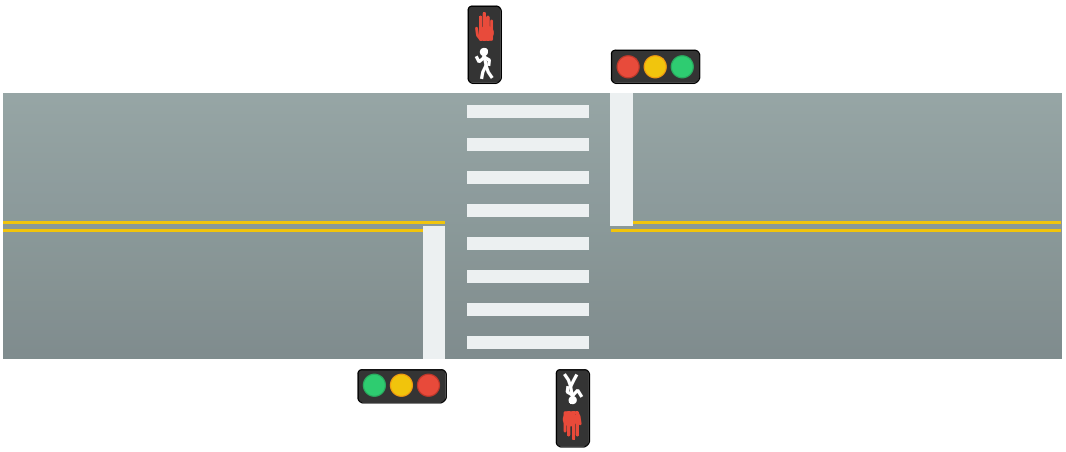

Figure 1: Schematic representation of the traffic light

To ensure the proper functioning of this intersection we want to verify properties such as "the light is never off", "the red light and green light cannot be lit at the same time", "cars should not be allowed in the intersection while the light for pedestrian is *Walk*", *etc.*

2.1.2 Modeling

We model this traffic light controller as a reactive system whose only input is the action performed on the button. The rest of the system evolves on its own and

responds to this (single) action. We use Boolean variables in Lustre to represent the lights, they are true when the corresponding light is on.

The traffic light for cars is modeled with these three Boolean variables:

- `Red`: true when the red light is on;
- `Yellow`: true when the yellow light is on;
- `Green`: true when the green light is on.

The traffic light for pedestrians is modeled with two Boolean variables:

- `Walk`: true when the *walk* sign is on;
- `DontWalk`: true when the *don't walk* sign is on.

The internal state of the controller is represented by an integer variable `Phase` which determines which lights should be lit or turned off.

The controller is modeled by a Lustre node `TrafficLight` which takes as input a Boolean saying if the button has been pressed in the current state and returns Boolean values which indicate which lights should be on or off.

```
node TrafficLight (Button : bool)
returns (Red, Yellow, Green, Walk, DontWalk : bool);
```

The internal state is handled by a local integer variable `Phase` (and another one which stores its value for technical reasons).

```
var Phase, prePhase : int;
```

In each step, the phase is computed as being `1` if the button is pressed or is incremented if it was less than `10`, and otherwise is reset to `0`.

```
prePhase = 0 -> pre Phase;
Phase    = if Button then 1
           else if prePhase > 0 and prePhase < 10
                then prePhase + 1
                else 0;
```

We decide which lights to turn on depending on the value of the phase. If it is `0`, the green light on the traffic sign is turned on, if it is greater than `1`, the red light is on, the *don't walk* light is on only if the `walk` light is off, *etc.*

```
Green    = Phase = 0;
Yellow   = Phase = 1;
Red      = Phase > 1;
```

```
Walk     = Phase > 2 and Phase < 10;
DontWalk = not Walk;
```

The complete Lustre code for the traffic light is given in the next section.

2.1.3 Specifying Critical Properties with Synchronous Observers

Lustre was designed from the ground-up to allow writing properties using the same constructs as the programming language. Every Boolean expression can be used to define a safety property (*i.e.* an invariant) of the system. The property is verified when the corresponding Boolean expression is true in *all the states* of the system.

To keep better track of names, we define properties by introducing special Boolean variables in the following of the section.

When using the traffic light controller, we can enforce certain constraints in order to specify the environmental assumptions. Here we just add an extra variable that says when cars are allowed to cross the intersection. This can vary depending on current regulations for instance. In our case we assume that cars are allowed when their light is either green or yellow.

```
CarsAllowed = Green or Yellow;
```

The most fundamental property that we want to guarantee is that there cannot be cars crossing the intersection when the light for pedestrians is set to *walk*.

```
R1 = not (CarsAllowed and Walk);
```

The other safety properties that we want ensure are the following:

- The traffic sign cannot have both lights red and green turned on at the same time.

  ```
  R2 = not (Red and Green);
  ```

- The traffic light cannot be off. There must be at least one light on at all times.

  ```
  R3 = Red or Yellow or Green;
  ```

- If the *walk* sign is on then the light for cars should be red.

```
R4 = Walk => Red;
```

- The light for pedestrian should always have one and only one light turned on.

```
R5 = Walk xor DontWalk;
```

- The traffic light cannot switch from red to green suddenly. It must go through the yellow light. The following property is true initially because there is no previous value for the status of the green light. When at least one step has elapsed, this ensures that the red light cannot be lit if the green light was on in the preceding step.

```
R6 = true -> not (Red and pre Green);
```

- A safety measure can be imposed to guarantee that no cars should be allowed in the intersection in the instant *preceding* the *walk* light being turned on.

```
R7 = true -> not (Walk and pre CarsAllowed);
```

- Similarly, we don't want pedestrians to be allowed in the intersection right before cars are allowed again.

```
R8 = true -> not (CarsAllowed and pre Walk);
```

The two previous properties enforce a safety delay between changes where no one is allowed in the intersection.

- This property guarantees that the yellow light cannot be on for more than one step.

```
R9 = true -> not (Yellow and pre Yellow);
```

- If the red light turned off (it was on in the previous state, but isn't anymore) then necessarily the green light must be on.

```
R10 = true -> (pre Red and not Red) => Green;
```

To avoid polluting the code for the Lustre program with specifications, an accepted practice is to write *synchronous observers* for properties.

These are programs that observe the input and output streams of the program S to be verified, and have a Boolean output stream for each safety property specified for S. The observers are defined so that, for each output stream, the output value is true at any point in the stream if, and only if, the corresponding safety property holds for S at that point. Note that synchronous observers are not limited to safety properties. For instance, they can be used to check more complex properties such as the equivalence of two programs. More in-depth explanations are available in the first two sections of [2].

The synchronous observer for the node `TrafficLight` is called `ReqTrafficLight`, it introduces local variables for *observing* the outputs of `TrafficLight` which it makes a call to. The special output Boolean variables are assigned their values and are declared as properties with the special annotation in comments.

We give in the following the Lustre program and specification for our traffic light system in its entirety.

```
--------------------------------------------------------------------------
-- This is an implementation of a simple traffic light system with one
-- (bi-directional) car lane and one pedestrian crossing.
--
--                             ooo
--       ------------------==--------------------
--                          ==
--                          ==
--       ------------------==--------------------
--                    ooo
--------------------------------------------------------------------------

node TrafficLight (Button : bool)
returns (Red, Yellow, Green, Walk, DontWalk : bool);

var Phase, prePhase : int;
let
  prePhase = 0 -> pre Phase;
  Phase    = if Button then 1
             else if prePhase > 0 and prePhase < 10
                  then prePhase + 1
                  else 0;

  Green    = Phase = 0;
  Yellow   = Phase = 1;
  Red      = Phase > 1;
```

```
  Walk     = Phase > 2 and Phase < 10;
  DontWalk = not Walk;
tel

--------------------------------------------------------------------------
-- A synchronous observer that checks a number of safety properties
-- for the TrafficLight.
--------------------------------------------------------------------------

node ReqTrafficLight (Button : bool)
returns (R1, R2, R3, R4, R5, R6, R7, R8, R9, R10: bool);

var CarsAllowed, Red, Yellow, Green, Walk, DontWalk : bool;
let
  (Red, Yellow, Green, Walk, DontWalk) = TrafficLight(Button);
  CarsAllowed = Green or Yellow;

  R1 = not (CarsAllowed and Walk);
  R2 = not (Red and Green);
  R3 = Red or Yellow or Green;
  R4 = Walk => Red;
  R5 = Walk xor DontWalk;
  R6 = true -> not (Red and pre Green);
  R7 = true -> not (Walk and pre CarsAllowed);
  R8 = true -> not (CarsAllowed and pre Walk);
  R9 = true -> not (Yellow and pre Yellow);
  R10 = true -> (pre Red and not Red) => Green;

  --%PROPERTY R1;
  --%PROPERTY R2;
  --%PROPERTY R3;
  --%PROPERTY R4;
  --%PROPERTY R5;
  --%PROPERTY R6;
  --%PROPERTY R7;
  --%PROPERTY R8;
  --%PROPERTY R9;
  --%PROPERTY R10;
tel
```

2.1.4 Verifying Safety Properties

Kind 2 can verify (most) safety properties entirely automatically. To verify our traffic light system, it suffices to pass the file name as argument to the `kind2` binary.

```
> kind2 TrafficLight.lus
...
--------------------------------------------------------------------------------
Summary_of_properties:

R10: invalid after 3 steps
R9: invalid after 2 steps
R8: invalid after 4 steps
R7: valid (at 2)
R6: valid (at 2)
R5: valid (at 1)
R4: valid (at 1)
R3: valid (at 1)
R2: valid (at 1)
R1: valid (at 1)
```

This output tells us that Kind 2 was able to verify properties `R1` to `R7` but found the properties `R8`, `R9`, and `R10` to be invalid. For those, it also provides a counterexample, *i.e.* values for the inputs (in our case this is a sequence of actions on the button) and the corresponding values of outputs and local variables that falsifies the property concerned. They are truncated from the output above and we give them in a nicer (but equivalent) format in Figure 2.

We can see that the property `R9` is violated, because we obtain a trace where the light is yellow twice in a row by pressing the button twice. We see that `R10` is falsifiable by pressing the button, waiting for one step, and pressing the button again. In this scenario the red light turns off but the yellow light is lit instead of the green as specified. The counterexample to `R8` is related to the last one and appears when the *walk* light is turned on and the button is pressed in the next step. The traffic lights then turns yellow which allows cars in the intersection. In this scenario, the delay is too short between the *walk* light and the yellow light.

2.1.5 Corrected Program

There are several ways to modify the program so that is respect the properties we have written. One such way is to make `Phase`'s values cycles between `0` and `10` with the exception that it can stay at `0` several steps in a row. This means that the traffic lights stays green until a pedestrian pushes the button, at which points it becomes yellow, then red, and after one step the *walk* light turns on and stays that way for six more steps, then the *don't walk* sign turns on, and after one extra step the traffic light goes back to green and remains green until another press on the button is registered.

To correct our program so that it has this behavior, we only have to modify the first condition of the update for the variable `Phase`. Now we set it to one also if its previous value was `0`. This means that all other button presses will be ignored.

Counterexample for R8:

Node ReqTrafficLight				
Inputs				
Button	1	0	0	1
Outputs				
R8	1	1	1	0
Locals				
CarsAllowed	1	0	0	1
Yellow	1	0	0	1
Green	0	0	0	0
Walk	0	0	1	0

Node TrafficLight				
Inputs				
Button	1	0	0	1
Outputs				
Red	0	1	1	0
Yellow	1	0	0	1
Green	0	0	0	0
Walk	0	0	1	0
DontWalk	1	1	0	1
Locals				
Phase	1	2	3	1

Counterexample for R9:

Node ReqTrafficLight		
Inputs		
Button	1	1
Outputs		
R9	1	0
Locals		
Yellow	1	1

Node TrafficLight		
Inputs		
Button	1	1
Outputs		
Red	0	0
Yellow	1	1
Green	0	0
Walk	0	0
DontWalk	1	1
Locals		
Phase	1	1

Counterexample for R10:

Node ReqTrafficLight			
Inputs			
Button	1	0	1
Outputs			
R10	1	1	0
Locals			
Red	0	1	0
Green	0	0	0

Node TrafficLight			
Inputs			
Button	1	0	1
Outputs			
Red	0	1	0
Yellow	1	0	1
Green	0	0	0
Walk	0	0	0
DontWalk	1	1	1
Locals			
Phase	1	2	1

Figure 2: Counterexample traces for the traffic light system

```
Phase    = if Button and prePhase = 0 then 1
           else if prePhase > 0 and prePhase < 10
                then prePhase + 1
                else 0;
```

We can also add an extra property `Cycling` in the body of the node `TrafficLight` to ensure that the resulting program has indeed the expected behavior. Note that the extra Boolean variable `Cycling` has to be declared in the node, another possibility is to inline the definition of `Cycling` after `--%PROPERTY`.

```
Cycling = true -> Phase <> 0 => Phase > pre Phase;
--%PROPERTY Cycling ;
```

Now, running Kind 2 on the modified file, we obtain the following which tells us that all properties have been verified.

```
> kind2 TrafficLight.lus
...
------------------------------------------------------------------------------
Summary_of_properties:

R10: valid (at 2)
R9: valid (at 2)
R8: valid (at 2)
R7: valid (at 2)
R6: valid (at 2)
R5: valid (at 2)
R4: valid (at 2)
R3: valid (at 2)
R2: valid (at 2)
R1: valid (at 2)
TrafficLight[l45c41].Cycling: valid (at 2)
```

The property named `TrafficLight[l45c41].Cycling` is a unique identifier corresponding to the property `Cycling` that we added for the node `TrafficLight`. Properties of subnodes are prefixed by the location of their call (here line 45, column 41).

3 Architecture

Kind 2 is organized in two major parts as depicted in Figure 3. The first one, the *frontend*, translates an input in the Lustre language to an internal first-order representation of the transition system. The second one is the heart of the model checker and consists in a supervisor that exchanges invariants between different engines running in parallel. This part reasons on the internal first-order representation and produces counter-examples or proofs certificates to be checked after the fact.

3.1 Frontend

The frontend manipulates three intermediate representations of the original Lustre program. The first pass is a simple parsing phase where the textual file is represented internally by an abstract syntax tree (Lustre AST). This is then type checked and simplified to obtain a nomalized Lustre AST (Simplified AST on Figure 3). This is the intermediate representation level at which slicing works, conservatively removing

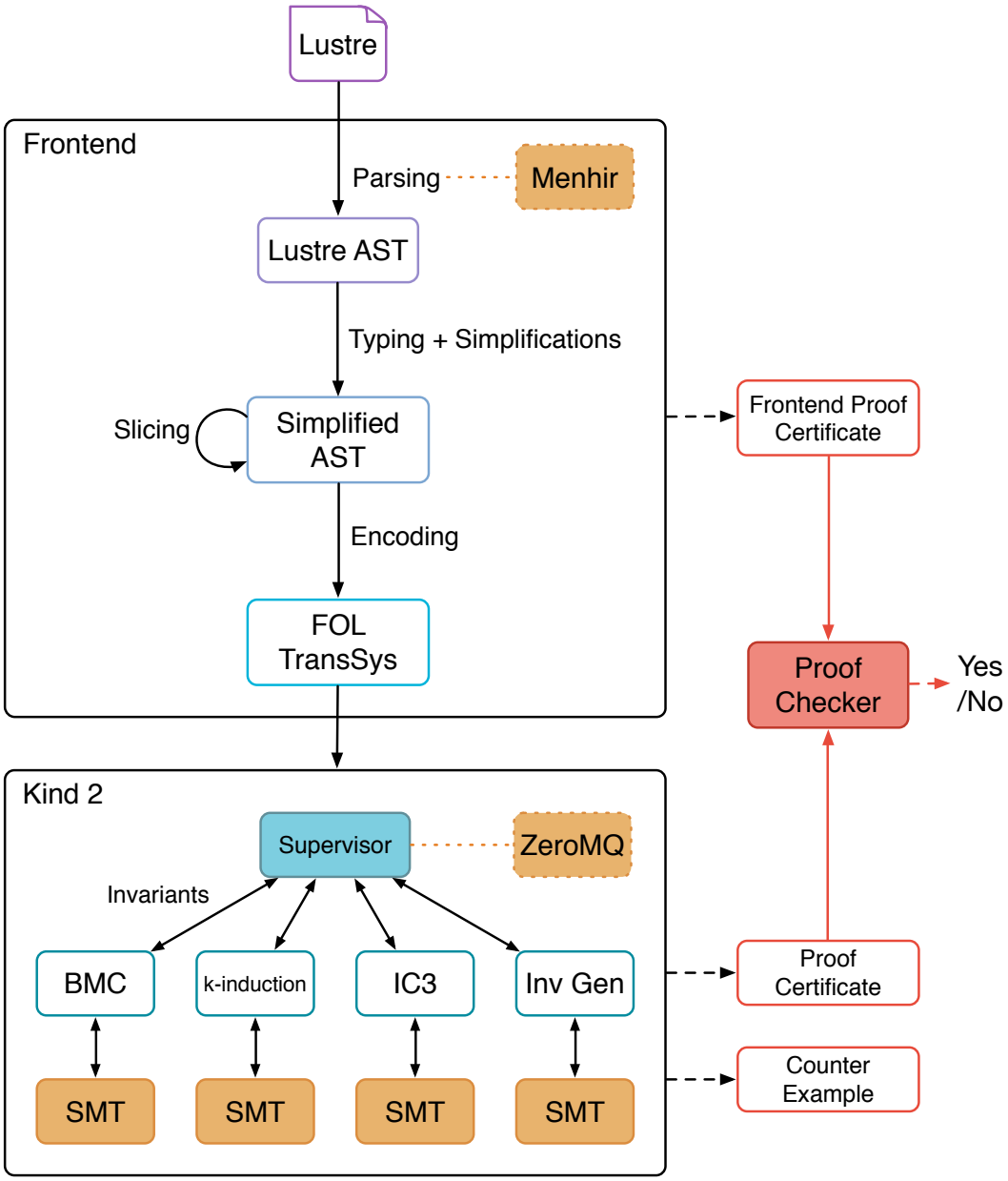

Figure 3: Architecture of Kind 2

parts of the system that do not impact the properties to be verified. Finally a simple one-to-one translation pass is applied, eliminating and encoding certain constructions to obtain a representation of the system in a fragment of first-order logic.

3.2 Engines

Kind 2's different model checking engines are managed by a supervisor which gathers and sends messages to the parallel processes with the message passing library ZeroMQ [**?**].

One important kind of messages is the one that contain newly discovered invariants. They are communicated by some engine to the supervisor which then tells the other engines of the extra knowledge. All these engines use one or several instances of SMT solvers (represented in filled boxed in Figure 3) as backend reasoning engines.

BMC

Bounded Model Checking or BMC is an engine that performs model checking by iteratively augmenting a bound k at which it looks for counter-examples. This technique is only able to find violations of properties but it can teach other engines that properties are true up to some bound. ★ citation

k-Induction

k-induction is an engine that checks properties to be preserved by (an increasingly larger) k steps of the transition relation. Together with the knowledge of BMC, it can *prove* properties to be invariant of the system. ★ citation

IC3

IC3 is an engine that implements the recent model checking algorithm IC3 (Indubitable Correctness for the Iterative Construction of Inductive Clauses) by Bradley *et al.* [**?**]. A set of clauses is successively strengthened until a inductive invariant is obtained. ★ citation

Invariant Generation

Two invariant generation modules run in parallel of all the other ones. The first one looks for invariants that relate variables of the state, the other one looks for invariants that relate state variables and their *previous values*. Candidates invariants are mined from different sources with a graph base algorithm [] and proven by an internal k-induction engine. ★ citation

3.3 Proofs and Checker

Kind 2 produces two kinds of proof certificates.

Property proof certificates of simply proof certificates are proofs of the k-inductiveness of the property. The certificate is first produced as an SMT2 file and a solver (CVC4) then produces in turn a proof certificate in the LFSC language.

Frontend certificates are lightweight and constructed with the objective of keeping the whole chain entirely automatic.

Instead of proving semantics preservation between the Lustre input and the internal FOL representation of the transition relation, we prove the observational equivalence of two internal representations independently obtained from the same Lustre input.

4 Syntax

This section describes the syntax used by Kind 2. It is essentially a subset of Lustre V4 [3] with special annotations in comments. Because the specification only appears inside comments, Kind 2's input files can be compiled and understood by traditional Lustre compilers (such as Scade [1]).

Source files concerned by this section:

- `src/lustreLexer.mll` contains the lexical analyzer (*lexer*) in the ocamllex format;
- `src/lustreParser.mly` contains the syntax analyzer (parser) defined in the menhir format as well as the definitions of the *tokens* produced by the lexer;
- `src/lustreAst.mli` contains the interface for the *abstract syntax tree* (AST) produced by the parser;
- `src/lustreAst.ml` contains pretty-printing function for the AST;
- `src/lustreIdent.mli` contains the interface for the encoding of Lustre identifiers;
- `src/lustreIdent.ml` contains the data-structures and encoding algorithms for Lustre identifiers.

We use the following notations for grammars:

Notation	Description
$\langle rule \rangle^\star$	repetition of rule $\langle rule \rangle$ an arbitrary number of times including zero)
$\langle rule \rangle_t^\star$	repetition of rule $\langle rule \rangle$ an arbitrary number of times including zero), occurrences being separated by the terminal t
$\langle rule \rangle^+$	repetition of rule $\langle rule \rangle$ at least once
$\langle rule \rangle_t^+$	repetition of rule $\langle rule \rangle$ at least once, occurrences being separated by the terminal t
$\langle rule \rangle$?	optional usage of rule $\langle rule \rangle$ (i.e. zero or once)
($\langle rule \rangle$)	parenthesising; take caution not to confuse these parentheses with the terminals `(` and `)`

4.1 Lexical Analysis

The lexer is generated with ocamllex, it allows include files and discards comments.

The character sequence `include "FILE"`, where `FILE` is a relative or absolute path, is replaced with the contents of `FILE`. The include directive may appear at any position in the input. Recursive includes are allowed. Positions are labeled by line, column and file name.

Spaces, tabulations, carriage-returns, and new lines constitute the blank characters.The token `--` is a comment, everything following the token until the end of the line is discarded, unless the comment is a special comment, see below. Multi-line comments can be enclosed in `/*` and `*/`, or `(*` and `*)`.

The following special comments are parsed:

- `--%MAIN` marks the node containing the annotation as the top node. Only one annotation is allowed in the whole input. The rest of the line is ignored, in particular it does not matter whether the comment is terminated by a `;` or not.
- `--%PROPERTY` must be followed by a Lustre expression and a `;`, it marks a proof obligation for the node that contains the comment.
- `--@contract`, `--@require` and `--@ensure` are annotations for contracts.

Identifiers obey the following regular expression ⟨*ident*⟩:

$$\langle ident \rangle \quad ::= \quad (\texttt{a–z} \mid \texttt{A–Z} \mid \texttt{\_}\ )\ (\texttt{a–z} \mid \texttt{A–Z} \mid \texttt{0–9} \mid \texttt{\_}\ )^{\star}$$

Numeral constants obey the following regular expression ⟨*numeral*⟩:

$$\langle numeral \rangle \quad ::= \quad (\texttt{0–9})^{+}$$

Decimal constants obey the following regular expression ⟨*decimal_exponent*⟩:

$$\begin{array}{lll} \langle decimal \rangle & ::= & (\texttt{0–9})^{+}\texttt{.}\ (\texttt{0–9})^{+}(\texttt{E}\ (\texttt{+} \mid \texttt{-})?\ (\texttt{0–9})^{+})? \\ \langle exponent \rangle & ::= & (\texttt{0–9})^{+}\texttt{E}\ (\texttt{+} \mid \texttt{-})?\ (\texttt{0–9})^{+} \\ \langle decimal\_exponent \rangle & ::= & \langle decimal \rangle \mid \langle exponent \rangle \end{array}$$

The following identifiers are the keywords of the language:

`and`	`array`	`assert`	`bool`
`condact`	`const`	`current`	`div`
`else`	`enum`	`false`	`fby`
`function`	`if`	`int`	`let`
`mod`	`node`	`not`	`of`
`or`	`pre`	`real`	`returns`
`struct`	`subrange`	`tel`	`then`
`true`	`type`	`var`	`when`
`with`	`xor`		

4.2 Syntactic Analysis

The parser is generated with menhir [4]. It parses a Lustre program into an abstract syntax tree that is almost identical to the actual abstract syntax of the Lustre. At the top level, the abstract syntax tree is a list of declarations of nodes, constants and types.

The grammar of Kind 2's Lustre input files are given in Figure 4 and Figure 5. The entry point is the non-terminal $\langle main \rangle$.

The associativity and precedence of the operators and constructors of the language are given by the table in Figure 6, from the lowest to highest precedence. Operators with the same precedence are displayed on one line.

4.3 Unsupported Features

The parser tries to cover most of the features of Lustre V4. However some of the features described in the previous sections are parsed but not supported by Kind 2. Their use will not produce parsing error messages, but rather some other specific messages which are detailed in this section.

4.3.1 Array Slices

Array slices are not implemented, the following expression will be rejected.

```
a[1..3]
```

All applications of the production rule $\langle ident \rangle$ `[` $(\langle expr \rangle$ `..` $\langle expr \rangle)^{+}_{,}$ `]` for the non-terminal $\langle expr \rangle$ of Figure 5 will make Kind 2 print the error message "`Array slices not implemented`" on its output and exit.

4.3.2 Array Concatenation

Array concatenation is not implemented, the following expression will be rejected.

```
a1 | a2
```

All applications of the production rule $\langle expr \rangle$ `|` $\langle expr \rangle$ for the non-terminal $\langle expr \rangle$ of Figure 5 will make Kind 2 print the error message "`Array concatenation not implemented`" on its output and exit.

⟨*main*⟩ ::= ⟨*decl*⟩*
⟨*decl*⟩ ::= ⟨*const_decl*⟩ | ⟨*type_decl*⟩ | ⟨*node_decl*⟩ | ⟨*func_decl*⟩
| ⟨*node_param_inst*⟩

⟨*const_decl*⟩ ::= ⟨*ident*⟩$^+_,$ `:` ⟨*lustre_type*⟩ `;`
| (⟨*ident*⟩ | ⟨*typed_ident*⟩) `=` ⟨*expr*⟩ `;`
⟨*type_decl*⟩ ::= `type` ⟨*ident*⟩$^+_,$ `;`
| `type` ⟨*ident*⟩$^+_,$ `=` ⟨*lustre_type*⟩ `;`
| `type` ⟨*ident*⟩$^+_,$ `=` `struct`? `{` ⟨*typed_idents*⟩$^\star_;$ `}` `;`
⟨*func_decl*⟩ ::= `function` ⟨*ident*⟩ `(` ⟨*typed_idents*⟩$^\star_;$ `)`
`returns` `(` ⟨*typed_idents*⟩$^\star_;$ `)` `;`
⟨*node_decl*⟩ ::= `node` ⟨*ident*⟩ (`<<`(`type` ⟨*ident*⟩)$^\star_;$`>>`)? `(` ⟨*clc_typed_idents*⟩$^\star_;$ `)`
`returns` `(` ⟨*cl_typed_idents*⟩ `)` `;`
⟨*contract*⟩*(⟨*const_decl*⟩ | ⟨*var_decl*⟩)*
`let` ⟨*equation*⟩* `tel` (`.` | `;`)?
⟨*node_param_inst*⟩ ::= `node` ⟨*ident*⟩ `=` ⟨*ident*⟩ `(` ⟨*lustre_type*⟩ `)` `;`

⟨*var_decl*⟩ ::= `var` (⟨*cl_typed_idents*⟩ `;`)$^+$
⟨*clc_typed_idents*⟩ ::= `const`? ⟨*typed_idents*⟩
| `const`? ⟨*typed_idents*⟩ `when` ⟨*clock*⟩
| `(` (`const`? ⟨*typed_idents*⟩)$^+_;$ `)` `when` ⟨*clock*⟩
⟨*cl_typed_idents*⟩ ::= ⟨*typed_idents*⟩
| ⟨*typed_idents*⟩ `when` ⟨*clock*⟩
| `(` ⟨*typed_idents*⟩$^+_;$ `)` `when` ⟨*clock*⟩
⟨*typed_idents*⟩ ::= ⟨*ident*⟩$^+_,$ `:` ⟨*lustre_type*⟩
⟨*lustre_type*⟩ ::= `bool` | `int` | `real` | ⟨*ident*⟩
| `subrange` `[` ⟨*expr*⟩ `,` ⟨*expr*⟩ `]` `of` `int`
| `[` ⟨*lustre_type*⟩$^+_,$ `]`
| ⟨*lustre_type*⟩ `^` ⟨*expr*⟩
| `enum` `{` ⟨*ident*⟩$^+_,$ `}`

⟨*contract*⟩ ::= `--@requires` ⟨*expr*⟩ `;` | `--@ensures` ⟨*expr*⟩ `;`
⟨*equation*⟩ ::= `assert` ⟨*expr*⟩ `;`
| (⟨*struct_item*⟩$^+$| `(` ⟨*struct_item*⟩*`)`) `=` ⟨*expr*⟩ `;`
| `--%MAIN` | `--%PROPERTY` ⟨*expr*⟩ `;`
⟨*clock*⟩ ::= ⟨*ident*⟩ | `not` ⟨*ident*⟩ | `not` `(` ⟨*ident*⟩ `)` | `true`

Figure 4: Grammar of Lustre files accepted by Kind 2

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle ident \rangle \mid \texttt{true} \mid \texttt{false} \mid \langle numeral \rangle \mid \langle decimal\_exponent \rangle \\
& \mid & \texttt{int}\ \langle expr \rangle \\
& \mid & \texttt{real}\ \langle expr \rangle \\
& \mid & \texttt{(}\ \langle expr \rangle^{+}_{,}\ \texttt{)} \\
& \mid & \texttt{[}\ \langle expr \rangle^{+}_{,}\ \texttt{]} \\
& \mid & \langle expr \rangle\ \texttt{\^{}}\ \langle expr \rangle \\
& \mid & \langle ident \rangle\ \texttt{[}\ \langle expr \rangle\ \texttt{]} \\
& \mid & \langle ident \rangle\ \texttt{[}\ (\langle expr \rangle\ \texttt{..}\ \langle expr \rangle)^{+}_{,}\ \texttt{]} \\
& \mid & \langle ident \rangle\ \texttt{.}\ \langle ident \rangle \\
& \mid & \langle ident \rangle\ \texttt{\{}\ (\langle ident \rangle\ \texttt{=}\ \langle expr \rangle)^{\star}_{;}\ \texttt{\}} \\
& \mid & \langle expr \rangle\ \texttt{|}\ \langle expr \rangle \\
& \mid & \langle expr \rangle\ \langle operator \rangle\ \langle expr \rangle \\
& \mid & (\texttt{-} \mid \texttt{not})\ \langle expr \rangle \\
& \mid & \texttt{\#(}\ \langle expr \rangle^{+}_{,}\ \texttt{)} \\
& \mid & \texttt{if}\ \langle expr \rangle\ \texttt{then}\ \langle expr \rangle\ \texttt{else}\ \langle expr \rangle \\
& \mid & \texttt{with}\ \langle expr \rangle\ \texttt{then}\ \langle expr \rangle\ \texttt{else}\ \langle expr \rangle \\
& \mid & \langle expr \rangle\ \texttt{when}\ \langle expr \rangle \\
& \mid & \texttt{current}\ \langle expr \rangle \\
& \mid & \texttt{condact (}\ \langle expr \rangle\ \texttt{,}\ \langle ident \rangle\ \texttt{(}\ \langle ident \rangle^{\star}_{,}\ \texttt{)}\ \texttt{,}\ \langle expr \rangle^{+}_{,}\ \texttt{)} \\
& \mid & \texttt{pre}\ \langle expr \rangle \\
& \mid & \texttt{fby (}\ \langle expr \rangle\ \texttt{,}\ \langle numeral \rangle\ \texttt{,}\ \langle expr \rangle\ \texttt{)} \\
& \mid & \langle ident \rangle\ \texttt{(}\ \langle expr \rangle^{\star}_{,}\ \texttt{)} \\
& \mid & \langle ident \rangle\ \texttt{<<}\ \langle lustre\_type \rangle^{\star}_{;}\ \texttt{>>}\ \texttt{(}\ \langle expr \rangle^{\star}_{,}\ \texttt{)} \\
\\
\langle operator \rangle & ::= & \texttt{-} \mid \texttt{+} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>} \\
& \mid & \texttt{div} \mid \texttt{not} \mid \texttt{mod} \mid \texttt{and} \mid \texttt{or} \mid \texttt{xor}
\end{array}
$$

Figure 5: Grammar of Lustre expressions accepted by Kind 2

operator or constructor	associativity
`|`	left
`else`	—
`->`	right
`=>`	right
`or xor`	left
`and`	left
`< <= = <> >= >`	left
`+ -`	left
`* div mod /`	left
`when`	—
`not`	—
`current`	—
`pre`	—
`^`	left
`int real`	left

Figure 6: Associativity and precedence of operators in Kind 2

4.3.3 Interpolation to base clock

Interpolation to the base clock is not implemented. Expressions with `current`, whether they contain `when` or not, like in the following, will be rejected.

```
current( f(x when clock) )
```

All applications of the production rule `current when` ⟨*expr*⟩ for the non-terminal ⟨*expr*⟩ of Figure 5 will make Kind 2 print the error message "`Current expression not supported`" on its output and exit.

All applications of the production rule `when` ⟨*expr*⟩ for the non-terminal ⟨*expr*⟩ of Figure 5 will make Kind 2 print the error message "`When expression must be the argument of a current operator`" on its output and exit.

`current` must be followed by `when` and Kind 2 will print the error message "`Current operator must have a when expression as argument`" on its output and exit if it is not the case.

4.3.4 One-hot

The Boolean *at-most one* n-ary operator (or one-hot) is not implemented, the following expression will be rejected.

```
#(a1, a2, a3, 0, 1)
```

All applications of the production rule **#(** $\langle expr \rangle^{+}_{,}$ **)** for the non-terminal $\langle expr \rangle$ of Figure 5 will make Kind 2 print the error message "`One-hot expression not supported`" on its output and exit.

4.3.5 Followed by

The *followed by* operator `fby` is not implemented, the following expression will be rejected.

```
1.0 fby cos
```

All applications of the production rule $\langle expr \rangle$ **|** $\langle expr \rangle$ for the non-terminal $\langle expr \rangle$ of Figure 5 will make Kind 2 print the error message "`Array concatenation not implemented`" on its output and exit.

4.3.6 Recursive node definitions

Recursive node calls are not supported, the following expression will be rejected.

```
node REC_DELAY (const d: int; X: bool) returns (Y: bool);
let
  Y = with d=0 then X
  else false -> pre(REC_DELAY(d-1, X));
tel
```

All applications of the production rule `with` $\langle expr \rangle$ `then` $\langle expr \rangle$ `else` $\langle expr \rangle$ for the non-terminal $\langle expr \rangle$ of Figure 5 will make Kind 2 print the error message "`Recursive nodes not supported`" on its output and exit.

4.3.7 Parametric nodes

Parametric nodes are not implemented, the following expressions will be rejected.

```
node mk_tab <<type t; const init: t; const size: int>> (a:t)
returns (res: t^size);
let
  res =  init ^ size;
tel
```

```
node tab_int3 = mk_tab<<int, 0, 3>>;
node param_node2 = mk_tab<<bool, true, 4>>;
```

All applications of the production rule `<<(type` $\langle ident \rangle)^{\star}_{;}$`>>` for the non-terminal $\langle node\_decl \rangle$ of Figure 4 will make Kind 2 print the error message "`Parametric nodes not supported`" on its output and exit.

All applications of the production rule $\langle ident \rangle$ `<<` $\langle lustre\_type \rangle^{\star}_{;}$ `>> (` $\langle expr \rangle^{\star}_{,}$ `)` for the non-terminal $\langle expr \rangle$ of Figure 5 will make Kind 2 print the error message "`Parametric nodes not supported`" on its output and exit.

All applications of the production rule for the non-terminal $\langle node\_param\_inst \rangle$ of Figure 4 will make Kind 2 print the error message "`Parametric nodes not supported`" on its output and exit.

4.3.8 Functions

Function declarations are not supported, the following expression will be rejected.

```
function succ (x:int) returns (int) ;
```

All applications of the production rule for the non-terminal $\langle func\_decl \rangle$ of Figure 4 will make Kind 2 print the error message "`Functions not supported`" on its output and exit.

4.3.9 Simplified Grammar

If we remove all the unsupported features from the grammar definition, we obtain the grammar of Figure 7 which can serve as a reference for users. The set of reserved identifiers for keywords remains unchanged.

4.4 Implementation Details

ASTs returned by the parser are decorated with position information that contains the file name as well as the line and column numbers:

```
type position =
  { pos_fname : string; pos_lnum: int; pos_cnum: int }
```

For instance, declarations are represented by the following type:

$\langle main \rangle$ ::= $\langle decl \rangle^\star$
$\langle decl \rangle$ ::= $\langle const\_decl \rangle \mid \langle type\_decl \rangle \mid \langle node\_decl \rangle$

$\langle const\_decl \rangle$ ::= $\langle ident \rangle^+_{,}$ `:` $\langle lustre\_type \rangle$ `;`
| $(\langle ident \rangle \mid \langle typed\_ident \rangle)$ `=` $\langle expr \rangle$ `;`
$\langle type\_decl \rangle$ ::= `type` $\langle ident \rangle^+_{,}$ `;`
| `type` $\langle ident \rangle^+_{,}$ `=` $\langle lustre\_type \rangle$ `;`
| `type` $\langle ident \rangle^+_{,}$ `=` `struct`? `{` $\langle typed\_idents \rangle^\star_{;}$ `}` `;`
$\langle node\_decl \rangle$ ::= `node` $\langle ident \rangle$ `(` (`const`? $\langle typed\_idents \rangle)^\star_{;}$ `)`
`returns` `(` $\langle typed\_idents \rangle$ `)` `;`
$\langle contract \rangle^\star (\langle const\_decl \rangle \mid \langle var\_decl \rangle)^\star$
`let` $\langle equation \rangle^\star$ `tel` (`.` | `;`)?

$\langle var\_decl \rangle$ ::= `var` $(\langle typed\_idents \rangle$ `;`$)^+$
$\langle typed\_idents \rangle$ ::= $\langle ident \rangle^+_{,}$ `:` $\langle lustre\_type \rangle$
$\langle lustre\_type \rangle$ ::= `bool` | `int` | `real` | $\langle ident \rangle$
| `subrange` `[` $\langle expr \rangle$ `,` $\langle expr \rangle$ `]` `of` `int`
| `[` $\langle lustre\_type \rangle^+_{,}$ `]`
| $\langle lustre\_type \rangle$ `^` $\langle expr \rangle$
| `enum` `{` $\langle ident \rangle^+_{,}$ `}`

$\langle contract \rangle$ ::= `--@requires` $\langle expr \rangle$ `;` | `--@ensures` $\langle expr \rangle$ `;`
$\langle equation \rangle$ ::= `assert` $\langle expr \rangle$ `;`
| $(\langle struct\_item \rangle^+ \mid$ `(` $\langle struct\_item \rangle^\star$ `)` $)$ `=` $\langle expr \rangle$ `;`
| `--%MAIN` | `--%PROPERTY` $\langle expr \rangle$ `;`

$\langle expr \rangle$ ::= $\langle ident \rangle$ | `true` | `false` | $\langle numeral \rangle$ | $\langle decimal\_exponent \rangle$
| `int` $\langle expr \rangle$
| `real` $\langle expr \rangle$
| `(` $\langle expr \rangle^+_{,}$ `)`
| `[` $\langle expr \rangle^+_{,}$ `]`
| $\langle expr \rangle$ `^` $\langle expr \rangle$
| $\langle ident \rangle$ `[` $\langle expr \rangle$ `]`
| $\langle ident \rangle$ `.` $\langle ident \rangle$
| $\langle ident \rangle$ `{` $(\langle ident \rangle$ `=` $\langle expr \rangle)^\star_{;}$ `}`
| $\langle expr \rangle$ $\langle operator \rangle$ $\langle expr \rangle$
| (`-` | `not`) $\langle expr \rangle$
| `if` $\langle expr \rangle$ `then` $\langle expr \rangle$ `else` $\langle expr \rangle$
| `condact` `(` $\langle expr \rangle$ `,` $\langle ident \rangle$ `(` $\langle ident \rangle^\star_{,}$ `)` `,` $\langle expr \rangle^+_{,}$ `)`
| `pre` $\langle expr \rangle$
| $\langle ident \rangle$ `(` $\langle expr \rangle^\star_{,}$ `)`

$\langle operator \rangle$ ::= `-` | `+` | `*` | `/` | `<` | `>` | `<=` | `>=` | `=` | `<>`
| `div` | `not` | `mod` | `and` | `or` | `xor`

Figure 7: Grammar of Lustre files accepted and supported by Kind 2

```
type declaration =
  | TypeDecl of Lib.position * type_decl
  | ConstDecl of Lib.position * const_decl
  | NodeDecl of Lib.position * node_decl
  | FuncDecl of Lib.position * func_decl
  | NodeParamInst of Lib.position * node_param_inst
```

5 Typing

This section presents the type checking algorithm used by Kind 2. The different ASTs manipulated by this phase are described in section 5.1. Section 5.2 describes the typing algorithm.

Source files concerned by this section:

- `src/lustreAst.mli` contains the interface for the *abstract syntax tree* (AST) produced by the parser;
- `src/lustreAst.ml` contains pretty-printing function for the AST;
- `src/lustreSimplify.mli` contains the interface for the typing (and simplification) of Lustre ASTs;
- `src/lustreSimplify.ml` contains the type checking (and simplification) algorithms for Lustre ASTs.

5.1 Abstract Syntax Tree

The parsing phase constructs abstract syntax trees which will then be given to the simplification and type checking algorithms. We describe these trees with the grammar defined in Figure 8. The production τ represents the abstract syntax for

$$
\begin{array}{lcl}
\tau & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{real} \mid [e;e] \mid \alpha \mid (\vec{\tau}) \mid \{\overrightarrow{\alpha : \tau}\} \mid \tau \texttt{\^{}} e \mid \vec{\alpha} \\
e & ::= & i \mid c \mid e \ op \ e \mid un \ e \mid N(\vec{e}) \mid i.i \mid i[e] \mid (\vec{e}) \mid e \texttt{\^{}} e \mid \tau\{\overrightarrow{i = e}\} \\
 & & \mid \texttt{ite}(e,e,e) \mid \texttt{to\_int}(e) \mid \texttt{to\_real}(e) \mid \texttt{condact}(e, N, \vec{e}, \vec{e}) \mid e \rightarrow e \\
s & ::= & i \mid i[e] \mid i.i \mid (\vec{s}) \\
q & ::= & s = e \mid \texttt{assert}(e) \mid \texttt{MAIN} \mid \texttt{PROPERTY} \ e \\
@ & ::= & \texttt{requires} \ e \mid \texttt{ensures} \ e \\
d & ::= & \texttt{type} \ \alpha \mid \texttt{const} \ i : \tau \mid \texttt{node} \ N(\overrightarrow{i : \tau}) \ \texttt{returns} \ (\overrightarrow{i : \tau}) \ \texttt{var} \ (\overrightarrow{i : \tau}) \ \vec{q} \ \vec{@}
\end{array}
$$

Figure 8: Lustre abstract syntax tree

the types of the language:

- the base types `bool`, `int` and `real`;

- the subrange types $[e; e]$;
- the tuples $(\vec{\tau})$, records $\{\overrightarrow{\alpha : \tau}\}$ and arrays $\tau \hat{} e$;
- and the types defined buy the user α.

The production e represents the expressions (both terms and formulas) of the language:

- i represents the identifiers;
- c can represent Boolean, integer or real constants;
- op regroups all binary operators of arithmetic (`+`,`*`,`/`, `div`, `mod`), comparison (`<`, `>`, `<=`, `>=`, `=`, `<>`) and propositional logic (`and`, `or`, `xor`);
- un represents the unary operators `not` and `-`;
- $N(\vec{e})$ is the node call of node N, the notation $\vec{e}$ stands for the arguments of the call;
- $i.i$ is the projection of a record;
- $i[e]$ is the projection of a tuple;
- $\tau\{\overrightarrow{i = e}\}$ is the notation for record literals (they are preceded by their type);
- $(\vec{e})$ is the notation for tuple (or array) literals;
- $e \hat{} e$ is the construction of a tuple with identical components or an array with identical values;
- the *if-then-else* construct is represented by the expression $\texttt{ite}(e, e, e)$;
- `to_int` (*resp.* `to_real`) represents the type coercion to integers (*resp.* reals);
- and we use $\rightarrow$ to represent *followed by*.

The left-hand side of the equations (q) are defined by the production s where i are identifiers. @ is the language of contracts and finally the production d defines the declarations of the language:

- `type` α represents the declaration of a type α;
- the user defined constants are defined by `const` $i : \tau$;
- and node definitions have a list of typed arguments and typed returned identifiers as well as local typed variable declaration. They are defined by a list of equations $\vec{q}$ and contracts $\vec{@}$.

5.2 Typing Algorithm

The typing algorithm is composed of five functions:

1. $\vdash_{Ty}$ verifies the well-formedness of type definitions and declarations;
2. $\vdash_{E}$ verifies and assigns types to expressions (and left-hand sides of equations);
3. $\vdash_{Q}$ verifies the types of equations;
4. $\vdash_{C}$ verifies that contracts are well-formed;
5. $\vdash_{D}$ verifies that top-level declarations are well-formed.

These functions manipulates the three following typing environments:

- the environment Θ contains bindings of the form $\alpha : \tau$ for declaration of type aliases introduced by `type`;

- the environment Δ contains bindings of the form $N : \vec{\tau} \to \vec{v}$ for nodes defined with declarations of the form `node` $N \ldots$;
- the environment Γ contains bindings $x : \tau$ for *global constants* declared with `const`, but also for *local variables* introduced by the `node` and `var` constructs.

To simplify notations, we write $A \vdash \vec{x}$ for $A \vdash x_1 \ \ldots \ A \vdash x_n$. We use the notation $x \notin B$ when B contains bindings to say that x is not bound in B. We write $\vec{x} \notin B$ to denote that neither x_1 is bound in B nor $\ldots$ nor x_n is bound in B. To add bindings, we use the notation $A + x : v$ for the environment obtained by adding a binding $x : v$ to A.

The underlying algorithms are defined by the following sequents:

1. $\Theta, \Delta, \Gamma \vdash_E e : \tau$ reads as "in the typing environment Θ, the definition environment Δ and the local environment Γ, the expression e is well-typed and its type is τ" (see Figure 9 and Figure 10 for typing rules with integer subranges).
2. $\Theta, \Delta, \Gamma \vdash_{Ty} \tau$ reads as "in the typing environment Θ, the definition environment Δ and the local environment Γ, the type τ is well-formed" (see Figure 11).
3. $\Theta, \Delta, \Gamma \vdash_Q q$ reads as "in the typing environment Θ, the definition environment Δ and the local environment Γ, the equation q is well-formed and well-typed" (see Figure 12).
4. $\Theta, \Delta, \Gamma \vdash_C @$ reads as "in the typing environment Θ, the definition environment Δ and the local environment Γ, the contract $@$ is well-formed and well-typed" (see Figure 13).
5. $\Theta, \Delta, \Gamma \vdash_D d : (\Theta', \Delta', \Gamma')$ reads as "in the typing environment Θ, the definition environment Δ and the local environment Γ, the declaration d is well-formed and produces a new typing environment Θ', a new definition environment Δ' and a new local environment Γ'" (see Figure 14).

5.3 Implementation Details

★ Talk about typing + simplifications done at the same time.

6 Lustre-to-Lustre simplifications

6.1 Simplified Abstract Syntax Tree

The simplified AST is defined by the grammar described in Figure 15.

Th expressions of this language (rule e) do not contain node calls, temporal operators or terms under a `pre` operator. An expression is always of the form $i \to t$ which gives the values in the initial state (term i) and the following states (term t).

The production rule τ represents the types which are almost the same as before excepted subranges are defined with integer constants and arrays are given a type for their indexes and a type for their values. Terms are free variables (x), bound

$$\textsc{Const}\ \frac{c \text{ is a constant of type } \tau \qquad \tau \in \{\texttt{bool}, \texttt{int}, \texttt{real}\}}{\Theta, \Delta, \Gamma \vdash_E c : \tau} \qquad \textsc{Var}\ \frac{x : \tau \in \Gamma}{\Theta, \Delta, \Gamma \vdash_E x : \tau}$$

$$\textsc{Arith}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : \tau \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : \tau \qquad op \in \{\texttt{+}, \texttt{-}, \texttt{/}, \texttt{*}\} \qquad \tau \in \{\texttt{int}, \texttt{real}\}}{\Theta, \Delta, \Gamma \vdash_E e_1\ op\ e_2 : \tau}$$

$$\textsc{Minus}\ \frac{\Theta, \Delta, \Gamma \vdash_E e : \tau \qquad \tau \in \{\texttt{int}, \texttt{real}\}}{\Theta, \Delta, \Gamma \vdash_E \texttt{-}\ e : \tau} \qquad \textsc{Not}\ \frac{\Theta, \Delta, \Gamma \vdash_E e : \texttt{bool}}{\Theta, \Delta, \Gamma \vdash_E \texttt{not}\ e : \texttt{bool}}$$

$$\textsc{IntOp}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : \texttt{int} \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : \texttt{int} \qquad op \in \{\texttt{div}, \texttt{mod}\}}{\Theta, \Delta, \Gamma \vdash_E e_1\ op\ e_2 : \texttt{int}}$$

$$\textsc{CompNum}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : \tau \quad \Theta, \Delta, \Gamma \vdash_E e_2 : \tau \quad op \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\} \quad \tau \in \{\texttt{int}, \texttt{real}, [l; u]\}}{\Theta, \Delta, \Gamma \vdash_E e_1\ op\ e_2 : \texttt{bool}}$$

$$\textsc{Eq}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : \tau \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : \tau}{\Theta, \Delta, \Gamma \vdash_E e_1 \texttt{ = } e_2 : \texttt{bool}} \qquad \textsc{Arrow}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : \tau \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : \tau}{\Theta, \Delta, \Gamma \vdash_E e_1 \to e_2 : \tau}$$

$$\textsc{Literal}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : \texttt{bool} \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : \texttt{bool} \qquad op \in \{\texttt{or}, \texttt{and}, \texttt{xor}\}}{\Theta, \Delta, \Gamma \vdash_E e_1\ op\ e_2 : \texttt{bool}}$$

$$\textsc{Call}\ \frac{N : (\tau_1, \ldots, \tau_n) \to (\upsilon_1, \ldots, \upsilon_m) \in \Delta \qquad \Theta, \Delta, \Gamma \vdash_E e_1 : \tau_1\ \ldots\ \Theta, \Delta, \Gamma \vdash_E e_n : \tau_n}{\Theta, \Delta, \Gamma \vdash_E N(e_1, \ldots, e_n) : (\upsilon_1, \ldots, \upsilon_m)}$$

$$\textsc{RecordProj}\ \frac{\Theta, \Delta, \Gamma \vdash_E r : \{\ldots; f : \tau; \ldots\}}{\Theta, \Delta, \Gamma \vdash_E r.f : \tau}$$

$$\textsc{TupleProj}\ \frac{\Theta, \Delta, \Gamma \vdash_E v : (\tau_1, \ldots, \tau_n) \qquad \Theta, \Delta, \Gamma \vdash_E i : \texttt{int} \qquad i \leq n}{\Theta, \Delta, \Gamma \vdash_E v[i] : \tau_i}$$

$$\textsc{Tuple}\ \frac{\Theta, \Delta, \Gamma \vdash_E v_1 : \tau_1\ \ldots\ \Theta, \Delta, \Gamma \vdash_E v_n : \tau_n}{\Theta, \Delta, \Gamma \vdash_E (v_1, \ldots, v_n) : (\tau_1, \ldots, \tau_n)} \qquad \textsc{Array}\ \frac{\Theta, \Delta, \Gamma \vdash_E v : \tau \qquad \Theta, \Delta, \Gamma \vdash_E n : \texttt{int}}{\Theta, \Delta, \Gamma \vdash_E v \hat{} n : (\tau, \ldots, t)}$$

$$\textsc{Record}\ \frac{t : \{f_1 : \tau_1; \ldots; f_n : \tau_n\} \in \Theta \qquad \Theta, \Delta, \Gamma \vdash_E v_1 : \tau_1\ \ldots\ \Theta, \Delta, \Gamma \vdash_E v_n : \tau_n}{\Theta, \Delta, \Gamma \vdash_E t\{f_1 = v_1; \ldots; f_n = v_n\} : \{f_1 : \tau_1; \ldots; f_n : \tau_n\}}$$

$$\textsc{Ite}\ \frac{\Theta, \Delta, \Gamma \vdash_E c : \texttt{bool} \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : \tau \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : \tau}{\Theta, \Delta, \Gamma \vdash_E \texttt{ite}(c, e_1, e_2) : \tau}$$

$$\textsc{ToInt}\ \frac{\Theta, \Delta, \Gamma \vdash_E e : \texttt{real}}{\Theta, \Delta, \Gamma \vdash_E \texttt{to\_int}(e) : \texttt{int}} \qquad \textsc{ToReal}\ \frac{\Theta, \Delta, \Gamma \vdash_E e : \texttt{int}}{\Theta, \Delta, \Gamma \vdash_E \texttt{to\_real}(e) : \texttt{real}}$$

$$\textsc{Condact}\ \frac{\begin{array}{c}\Theta, \Delta, \Gamma \vdash_E N(e_1, \ldots, e_n) : (\upsilon_1, \ldots, \upsilon_m) \\ \Theta, \Delta, \Gamma \vdash_E d_1 : \upsilon_1\ \ldots\ \Theta, \Delta, \Gamma \vdash_E d_m : \upsilon_m \qquad \Theta, \Delta, \Gamma \vdash_E c : \texttt{bool}\end{array}}{\Theta, \Delta, \Gamma \vdash_E \texttt{condact}(c, N, (e_1, \ldots, e_n), (d_1, \ldots, d_m)) : (\upsilon_1, \ldots, \upsilon_m)}$$

Figure 9: Typing of expressions

$$\textsc{SubPlus}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : [a, b] \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : [c, d]}{\Theta, \Delta, \Gamma \vdash_E e_1 \texttt{ + } e_2 : [a + c, b + d]}$$

$$\textsc{SubMinus}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : [a, b] \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : [c, d]}{\Theta, \Delta, \Gamma \vdash_E e_1 \texttt{ - } e_2 : [a - d, b - c]} \qquad \textsc{SubUminus}\ \frac{\Theta, \Delta, \Gamma \vdash_E e : [a, b]}{\Theta, \Delta, \Gamma \vdash_E \texttt{ - } e : [-b, -a]}$$

$$\textsc{SubMod}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : [a, b] \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : [c, d]}{\Theta, \Delta, \Gamma \vdash_E e_1\ \texttt{mod}\ e_2 : [0, max(|c|, |d|) - 1]}$$

$$\textsc{SubUndef}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : [a, b] \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : [c, d] \qquad op\ \in \{\texttt{div}, *\}}{\Theta, \Delta, \Gamma \vdash_E e_1\ op\ e_2 : \texttt{int}}$$

$$\textsc{Arrow}\ \frac{\Theta, \Delta, \Gamma \vdash_E e_1 : [a, b] \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : [c, d]}{\Theta, \Delta, \Gamma \vdash_E e_1 \rightarrow e_2 : [min(a, c), max(b, d)]}$$

$$\textsc{Ite}\ \frac{\Theta, \Delta, \Gamma \vdash_E c : \texttt{bool} \qquad \Theta, \Delta, \Gamma \vdash_E e_1 : [a, b] \qquad \Theta, \Delta, \Gamma \vdash_E e_2 : [c, d]}{\Theta, \Delta, \Gamma \vdash_E \texttt{ite}(c, e_1, e_2) : [min(a, c), max(b, d)]}$$

Figure 10: Special typing rules for expressions in subranges

$$\textsc{BultinType}\ \frac{\tau \in \{\texttt{bool}, \texttt{int}, \texttt{real}\}}{\Theta, \Delta, \Gamma \vdash_{Ty} \tau} \qquad \textsc{Subrange}\ \frac{\Theta, \Gamma \vdash_E l : \texttt{int} \qquad \Theta, \Gamma \vdash_E u : \texttt{int}}{\Theta, \Delta, \Gamma \vdash_{Ty} [l; u]}$$

$$\textsc{UserType}\ \frac{\alpha : t \in \Theta \qquad \Theta \vdash_{Ty} t}{\Theta, \Delta, \Gamma \vdash_{Ty} \alpha} \qquad \textsc{RecordType}\ \frac{\Theta, \Delta, \Gamma \vdash_{Ty} t_1 \quad \ldots \quad \Theta, \Delta, \Gamma \vdash_{Ty} t_n}{\Theta, \Delta, \Gamma \vdash_{Ty} \{\alpha_1 : t_1; \ldots; \alpha_n : t_n\}}$$

$$\textsc{TupleType}\ \frac{\Theta, \Delta, \Gamma \vdash_{Ty} t_1 \quad \ldots \quad \Theta, \Delta, \Gamma \vdash_{Ty} t_n}{\Theta, \Delta, \Gamma \vdash_{Ty} (t_1, \ldots, t_n)}$$

$$\textsc{ArrayType}\ \frac{\Theta, \Delta, \Gamma \vdash_{Ty} t \qquad \Theta, \Gamma \vdash_E s : \texttt{int}}{\Theta, \Delta, \Gamma \vdash_{Ty} t\,\hat{}\,s}$$

Figure 11: Verification of types well-formedness

$$\textsc{Eqeq}\ \frac{\Theta, \Delta, \Gamma \vdash_E s : \tau \qquad \Theta, \Delta, \Gamma \vdash_E e : \tau}{\Theta, \Delta, \Gamma \vdash_Q s = e} \qquad \textsc{Assert}\ \frac{\Theta, \Delta, \Gamma \vdash_E e : \texttt{bool}}{\Theta, \Delta, \Gamma \vdash_Q \texttt{assert}(e)}$$

$$\textsc{Main}\ \frac{}{\Theta, \Delta, \Gamma \vdash_Q \texttt{MAIN}} \qquad \textsc{Prop}\ \frac{\Theta, \Delta, \Gamma \vdash_E e : \texttt{bool}}{\Theta, \Delta, \Gamma \vdash_Q \texttt{PROPERTY}\ e}$$

Figure 12: Typing of equations

$$\textsc{Require}\ \frac{\Theta, \Delta, \Gamma \vdash_E e : \texttt{bool}}{\Theta, \Delta, \Gamma \vdash_C \texttt{@requires}\ e} \qquad \textsc{Ensures}\ \frac{\Theta, \Delta, \Gamma \vdash_E e : \texttt{bool}}{\Theta, \Delta, \Gamma \vdash_C \texttt{@ensures}\ e}$$

Figure 13: Typing of contracts

$$\textsc{TypeDecl}\ \frac{t \notin \Theta \qquad \Theta \vdash_{Ty} \tau}{\Theta, \Delta, \Gamma \vdash_D \texttt{type}\ t = \tau : (\Theta + t : \tau, \Gamma)}$$

$$\textsc{ConstDecl}\ \frac{c \notin \Gamma \qquad \Theta, \Gamma \vdash_E e : \tau}{\Theta, \Delta, \Gamma \vdash_D \texttt{const}\ c = e : (\Theta, \Delta, \Gamma + c : \tau)}$$

$$\textsc{NodeDecl}\ \frac{\begin{array}{c}\vec{i} \notin \Gamma \quad \vec{o} \notin \Gamma \quad \vec{v} \notin \Gamma \qquad \vec{i} + \vec{o} + \vec{v}\ \text{disjoint} \qquad \Theta \vdash_{Ty} \vec{\tau_i} \quad \Theta \vdash_{Ty} \vec{\tau_o} \quad \Theta \vdash_{Ty} \vec{\tau_v} \\ \Gamma' = \Gamma + \vec{i} : \vec{\tau_i} + \vec{o} : \vec{\tau_o} + \vec{v} : \vec{\tau_v} \qquad \Theta, \Delta, \Gamma' \vdash_Q \vec{q} \qquad \Theta, \Delta, \Gamma' \vdash_C \vec{@}\end{array}}{\Theta, \Delta, \Gamma \vdash_D \texttt{node}\ N(\vec{i} : \vec{\tau_i})\ \texttt{returns}\ (\vec{o} : \vec{\tau_o})\ \texttt{var}\ (\vec{v} : \vec{\tau_v})\ \vec{q}\ \vec{@} : (\Theta, \Delta + N : \vec{\tau_i} \to \vec{\tau_o}, \Gamma)}$$

Figure 14: Typing of declarations

variables ($\%n$ represented by DeBruijn indexes), symbols (f), applications ($f(\vec{t})$), let constructions and quantified (universally and existentially) terms. The expressions of the languages (e) are reduced to a term *followed by* a term. The production l represents the nodes of the file, they have a name (N) and variables as arguments which are decorated by a set of markers (m saying if the variable is an input of the node, an output, or a local variable as well as if it is a Boolean property, an oracle or an observer). These nodes are annotated with assertions (first component), contracts (ensures and requires, second component), and a Boolean marker for the *main* node (third component). Each node is defined by its set of equations (of the form $v = e$).

The transformations described in this section take care of turning a Lustre AST of Figure 8 into a simplified AST of Figure 15. During this phase, a number of simplification and transformation steps are taken. We list, explain and justify them in this section but we do not give formal rules for them in an effort to keep the

$$\begin{array}{lcl}
\tau & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{real} \mid [n; n] \mid \texttt{array}(\tau, \tau) \\
t & ::= & x \mid x\texttt{@}n \mid \%n \mid f \mid f(\vec{t}) \mid \texttt{let}\ \vec{\tau} = \vec{t}\ \texttt{in}\ t \mid \forall \vec{\tau}.\ t \mid \exists \vec{\tau}.\ t \\
e & ::= & t \to t : \tau \\
c & ::= & \vec{v} = N(c, \vec{v}, \vec{e})\ \textsf{obs}\ \vec{v} \\
m & ::= & \textsf{in} \mid \textsf{out} \mid \textsf{local} \mid \textsf{prop} \mid \textsf{oracle} \mid \textsf{obs} \\
l & ::= & N(\overrightarrow{v_m})\ [\vec{e}, (\vec{e}, \vec{e}), \vec{c}, \%b]\ \overrightarrow{v = e}
\end{array}$$

Figure 15: Simplified Lustre abstract syntax tree

document readable.

6.2 State Variables and pre

State variables or streams in Lustre are represented by special variables in this stage. These variables always appear with a suffix `@`n which denote the value of the stream at the instant n. A value of n is chosen to represent the current instant (let's say 0). The state variable x is then represented by what we call a *state variable instance* x`@`0. Because the simplification pushes `pre`, they are guaranteed to only appear over variables, like `pre` x and are instead replaced by a state variable instance of the previous instant (with respect to the current instant). In our example, `pre` x will be represented by x`@`-1.

6.3 Node Calls

7 Outputs

7.1 Results

$$
\begin{array}{lcl}
\mathit{stream} & ::= & \eta\ (\mathsf{in} \mid \mathsf{out} \mid \mathsf{local})\ \overrightarrow{(\kappa, v)} \\
\mathit{node} & ::= & \eta\ (\mathsf{line}\ \kappa, \mathsf{col}\ \kappa)\ \overrightarrow{\mathit{stream}} \\
\mathit{cex} & ::= & \overrightarrow{\mathit{node}} \\
\mathit{source} & ::= & \mathsf{IC3} \mid \mathsf{BMC} \mid \mathsf{IND} \mid \mathsf{Invgen} \mid \mathsf{InvgenOS} \\
\mathit{status} & ::= & \mathsf{valid}(\mathit{source}) \mid \mathsf{falsifiable}(\mathit{source}, \kappa, \mathit{cex}) \mid \mathsf{unknown}(\kappa?) \\
\mathit{prop} & ::= & \eta\ \mathit{status}\ \rho \\
\mathit{result} & ::= & \overrightarrow{\mathit{prop}}
\end{array}
$$

Figure 16: Abstract syntax tree for Kind 2's results output

In this AST definition we use the following symbols:

- η is a character string, used to represent names of streams, nodes and properties;
- κ stands for a positive integer value;
- ρ is a time quantity;
- v is used to denote values of streams; the type of v depends on the type of the stream;
- and we use ? to denote an optional component.

Each run of Kind 2 outputs results (production rule *result*) which associates properties to statuses (production rule *prop*). Properties are given with their status and the runtime (ρ) necessary to reach this conclusion.

When a property is valid or falsifiable, we give the *source* which made the claim (this can be IC3, BMC for negative results, k-induction for positive results or a form of invariant generation). Properties can also be unknown, which means that Kind 2 was not able to prove or disprove it. Sometimes, even though the property is unknown, Kind 2 is able to give a bound up-to which the property is true (this is the κ as optional argument of the constructor unknown).

Falsifiable properties come with a counterexample *cex* (ant its length κ). Counterexamples assign values to streams of certain nodes. Those are given a name and position (line, column in the source file). Finally streams (production *stream*) are simply given as the name of the Lustre variable (η), a marker to say if it is an input (in), output (out) or local (local). The essence of the stream lies in its last element which is a sequence which maps values (v) to instants (κ).

Example. *The output Kind 2 on the incorrect version of the traffic light system in section 2.1.4 in terms of the AST of figure 16 is:*

```
R1  valid(IND)                                                                  0.178 s
R2  valid(IND)                                                                  0.178 s
R3  valid(IND)                                                                  0.178 s
R4  valid(IND)                                                                  0.178 s
R5  valid(IND)                                                                  0.178 s
R6  valid(IND)                                                                  0.198 s
R7  valid(IND)                                                                  0.198 s
R8  falsifiable(BMC, 4,
    ReqTrafficLight
      Button        in     (0, true)   (1, false)  (2, false)  (3, true)
      R8            in     (0, true)   (1, true)   (2, true)   (3, false)
      CarsAllowed   local  (0, true)   (1, false)  (2, false)  (3, true)
      Yellow        local  (0, true)   (1, false)  (2, false)  (3, true)
      Green         local  (0, false)  (1, false)  (2, false)  (3, false)
      Walk          local  (0, false)  (1, false)  (2, true)   (3, false)
    TrafficLight (line 43, col 41)
      Button        in     (0, true)   (1, false)  (2, false)  (3, true)
      Red           out    (0, false)  (1, true)   (2, true)   (3, false)
      Yellow        out    (0, true)   (1, false)  (2, false)  (3, true)
      Green         out    (0, false)  (1, false)  (2, false)  (3, false)
      Walk          out    (0, false)  (1, false)  (2, true)   (3, false)
      DontWalk      out    (0, true)   (1, true)   (2, false)  (3, true)
      Phase         local  (0, 1)      (1, 2)     (2, 3)     (3, 1)     )  0.178 s
R9  falsifiable(IC3, 2,
```

	ReqTrafficLight						
	Button	in	(0, true)	(1, true)			
	R9	in	(0, true)	(1, false)			
	Yellow	local	(0, true)	(1, true)			
	TrafficLight (line 43, col 41)						
	Button	in	(0, true)	(1, true)			
	Red	out	(0, false)	(1, false)			
	Yellow	out	(0, true)	(1, true)			
	Green	out	(0, false)	(1, false)			
	Walk	out	(0, false)	(1, false)			
	DontWalk	out	(0, true)	(1, true)			
	Phase	local	(0, 1)	(1, 1)	)		0.158 s
R10	falsifiable(BMC, 2,						
	ReqTrafficLight						
	Button	in	(0, true)	(1, false)	(2, true)		
	R10	in	(0, true)	(1, true)	(2, false)		
	Red	local	(0, false)	(1, true)	(2, false)		
	Green	local	(0, false)	(1, false)	(2, false)		
	TrafficLight (line 43, col 41)						
	Button	in	(0, true)	(1, false)	(2, true)		
	Red	out	(0, false)	(1, true)	(2, false)		
	Yellow	out	(0, true)	(1, false)	(2, true)		
	Green	out	(0, false)	(1, false)	(2, false)		
	Walk	out	(0, false)	(1, false)	(2, false)		
	DontWalk	out	(0, true)	(1, true)	(2, true)		
	Phase	local	(0, 1)	(1, 2)	(2, 1)	)	0.178 s

7.1.1 Plain Text Output

```
kind2 v0.8.0-133-g7004523

<Failure> Property R9 is invalid by IC3 for k=2 after 0.158s.

Counterexample:
Node ReqTrafficLight ()
== Inputs ==
Button   true  true
== Outputs ==
R1       true  true
R2       true  true
R3       true  true
R4       true  true
R5       true  true
R6       true  true
R7       true  true
R8       true  true
```

```
R9       true  false
R10      true  true
== Locals ==
Yellow   true  true

Node TrafficLight (ReqTrafficLight[l43c41])
== Inputs ==
Button   true  true
== Outputs ==
Red      false false
Yellow   true  true
Green    false false
Walk     false false
DontWalk true  true
== Locals ==
Phase    1     1
prePhase 0     0


<Success> Property R1 is valid by inductive step after 0.178s.

<Success> Property R2 is valid by inductive step after 0.178s.

<Success> Property R3 is valid by inductive step after 0.178s.

<Success> Property R4 is valid by inductive step after 0.178s.

<Success> Property R5 is valid by inductive step after 0.178s.

<Failure> Property R10 is invalid by BMC for k=3 after 0.178s.

Counterexample:
Node ReqTrafficLight ()
== Inputs ==
Button   true  false true
== Outputs ==
R1       true  true  true
R2       true  true  true
R3       true  true  true
R4       true  true  true
R5       true  true  true
R6       true  true  true
R7       true  true  true
R8       true  true  true
R9       true  true  true
R10      true  true  false
```

```
== Locals ==
Red      false true  false
Green    false false false

Node TrafficLight (ReqTrafficLight[l43c41])
== Inputs ==
Button   true  false true
== Outputs ==
Red      false true  false
Yellow   true  false true
Green    false false false
Walk     false false false
DontWalk true  true  true
== Locals ==
Phase    1     2     1
prePhase 0     0     0


<Success> Property R7 is valid by inductive step after 0.198s.

<Success> Property R6 is valid by inductive step after 0.198s.

<Failure> Property R8 is invalid by BMC for k=4 after 0.198s.

Counterexample:
Node ReqTrafficLight ()
== Inputs ==
Button      true  false false true
== Outputs ==
R1          true  true  true  true
R2          true  true  true  true
R3          true  true  true  true
R4          true  true  true  true
R5          true  true  true  true
R6          true  true  true  true
R7          true  true  true  true
R8          true  true  true  false
R9          true  true  true  true
R10         true  true  true  false
== Locals ==
CarsAllowed true  false false true
Yellow      true  false false true
Green       false false false false
Walk        false false true  false

Node TrafficLight (ReqTrafficLight[l43c41])
```

```
== Inputs ==
Button      true  false false true
== Outputs ==
Red         false true  true  false
Yellow      true  false false true
Green       false false false false
Walk        false false true  false
DontWalk    true  true  false true
== Locals ==
Phase       1     2     3     1
prePhase    0     0     0     0


--------------------------------------------------------------------------------
Summary_of_properties:

R10: invalid after 3 steps
R9: invalid after 2 steps
R8: invalid after 4 steps
R7: valid (at 3)
R6: valid (at 3)
R5: valid (at 2)
R4: valid (at 2)
R3: valid (at 2)
R2: valid (at 2)
R1: valid (at 2)
```

7.1.2 XML Output

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- |===| Simple types. -->

  <!-- Log class. Among "info", "warn" and "fatal" -->
  <xs:simpleType name="logClass">
    <xs:restriction base="xs:string">
      <xs:enumeration value="info"/>
      <xs:enumeration value="warn"/>
      <xs:enumeration value="fatal"/>
    </xs:restriction>
  </xs:simpleType>
```

```
  <!-- Kind 2 module. Not restricted for now. -->
  <xs:simpleType name="k2Module">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>

  <!-- Time unit. Can only be "sec" for now. -->
  <xs:simpleType name="timeUnit">
    <xs:restriction base="xs:string">
      <xs:enumeration value="sec"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- Positive integer for k and instants. -->
  <xs:simpleType name="posInt">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- Stream type. Among "bool", "int" and "real". -->
  <xs:simpleType name="streamType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="bool"/>
      <xs:enumeration value="int"/>
      <xs:enumeration value="real"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- Stream class. Among "input", "output" and "local". -->
  <xs:simpleType name="streamClass">
    <xs:restriction base="xs:string">
      <xs:enumeration value="input"/>
      <xs:enumeration value="output"/>
      <xs:enumeration value="local"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- Property status. Among "valid" and "falsifiable". -->
  <xs:simpleType name="propStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="valid"/>
      <xs:enumeration value="falsifiable"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- |===| Elements. -->
```

```
<!-- Log. -->
<xs:element name="Log">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="class" type="logClass" use="required"/>
        <xs:attribute name="source" type="xs:string" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<!-- Runtime. -->
<xs:element name="Runtime">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:decimal">
        <xs:attribute name="unit" type="timeUnit" use="required"/>
        <xs:attribute name="timeout" type="xs:boolean" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<!-- Answer. -->
<xs:element name="Answer">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="propStatus">
        <xs:attribute name="source" type="k2Module" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<!-- All of the above is useless, we can't type the values of a trace. -->
<xs:simpleType name="genericValue">
  <xs:restriction base="xs:string">
    <!-- Bool. -->
    <xs:pattern value="false|true"/>
    <!-- Integer. -->
    <xs:pattern value="0|[1-9]([0-9])*"/>
    <!-- Real. -->
    <xs:pattern value="(0|[1-9]([0-9])*)\.(0|([0-9])*[1-9])"/>
    <xs:pattern value="(0|[1-9]([0-9])*)/(0|[1-9]([0-9])*)"/>
```

```
    </xs:restriction>
  </xs:simpleType>

  <!-- Value as output in cex's. -->
  <xs:element name="Value">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="genericValue">
          <xs:attribute name="instant" type="posInt" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <!-- Stream as output in cex's. -->
  <xs:element name="Stream">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Value" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="type" type="streamType" use="required"/>
      <xs:attribute name="class" type="streamClass" use="required"/>
    </xs:complexType>
  </xs:element>

  <!-- Node, as output in a cex. -->
  <xs:element name="Node">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Stream" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="Node" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="line" type="posInt"/>
      <xs:attribute name="column" type="posInt"/>
    </xs:complexType>
  </xs:element>

  <!-- Counter example. -->
  <xs:element name="Counterexample">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Node" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
```

```
  </xs:element>

  <!-- Property status. -->
  <xs:element name="Property">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Runtime" minOccurs="1" maxOccurs="1"/>
        <xs:element name="K" type="posInt" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="Answer" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="Counterexample" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>

  <!-- |===| Root. -->
  <xs:element name="Results">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="Log" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="Property" minOccurs="1" maxOccurs="unbounded"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

```
<?xml version="1.0"?>
<Results xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<Log class="info" source="parser">kind2 v0.8.0</Log>

<Property name="R1">
  <Runtime unit="sec" timeout="false">0.178</Runtime>
  <Answer source="indstep">valid</Answer>
</Property>

<Property name="R2">
  <Runtime unit="sec" timeout="false">0.178</Runtime>
  <Answer source="indstep">valid</Answer>
</Property>

<Property name="R3">
  <Runtime unit="sec" timeout="false">0.178</Runtime>
  <Answer source="indstep">valid</Answer>
```

```
</Property>

<Property name="R4">
  <Runtime unit="sec" timeout="false">0.178</Runtime>
  <Answer source="indstep">valid</Answer>
</Property>

<Property name="R5">
  <Runtime unit="sec" timeout="false">0.178</Runtime>
  <Answer source="indstep">valid</Answer>
</Property>

<Property name="R6">
  <Runtime unit="sec" timeout="false">0.198</Runtime>
  <Answer source="indstep">valid</Answer>
</Property>

<Property name="R7">
  <Runtime unit="sec" timeout="false">0.198</Runtime>
  <Answer source="indstep">valid</Answer>
</Property>

<Property name="R8">
  <Runtime unit="sec" timeout="false">0.178</Runtime>
  <K>4</K>
  <Answer source="bmc">falsifiable</Answer>
  <Counterexample>
    <Node name="ReqTrafficLight" >
      <Stream name="Button" type="bool" class="input">
        <Value instant="0">true</Value>
        <Value instant="1">false</Value>
        <Value instant="2">false</Value>
        <Value instant="3">true</Value>
      </Stream>
      <Stream name="R8" type="bool" class="output">
        <Value instant="0">true</Value>
        <Value instant="1">true</Value>
        <Value instant="2">true</Value>
        <Value instant="3">false</Value>
      </Stream>
      <Stream name="CarsAllowed" type="bool" class="local">
        <Value instant="0">true</Value>
        <Value instant="1">false</Value>
        <Value instant="2">false</Value>
        <Value instant="3">true</Value>
      </Stream>
```

```
      <Stream name="Yellow" type="bool" class="local">
        <Value instant="0">true</Value>
        <Value instant="1">false</Value>
        <Value instant="2">false</Value>
        <Value instant="3">true</Value>
      </Stream>
      <Stream name="Green" type="bool" class="local">
        <Value instant="0">false</Value>
        <Value instant="1">false</Value>
        <Value instant="2">false</Value>
        <Value instant="3">false</Value>
      </Stream>
      <Stream name="Walk" type="bool" class="local">
        <Value instant="0">false</Value>
        <Value instant="1">false</Value>
        <Value instant="2">true</Value>
        <Value instant="3">false</Value>
      </Stream>
      <Node name="TrafficLight" line="43" column="41">
        <Stream name="Button" type="bool" class="input">
          <Value instant="0">true</Value>
          <Value instant="1">false</Value>
          <Value instant="2">false</Value>
          <Value instant="3">true</Value>
        </Stream>
        <Stream name="Red" type="bool" class="output">
          <Value instant="0">false</Value>
          <Value instant="1">true</Value>
          <Value instant="2">true</Value>
          <Value instant="3">false</Value>
        </Stream>
        <Stream name="Yellow" type="bool" class="output">
          <Value instant="0">true</Value>
          <Value instant="1">false</Value>
          <Value instant="2">false</Value>
          <Value instant="3">true</Value>
        </Stream>
        <Stream name="Green" type="bool" class="output">
          <Value instant="0">false</Value>
          <Value instant="1">false</Value>
          <Value instant="2">false</Value>
          <Value instant="3">false</Value>
        </Stream>
        <Stream name="Walk" type="bool" class="output">
          <Value instant="0">false</Value>
          <Value instant="1">false</Value>
```

```
          <Value instant="2">true</Value>
          <Value instant="3">false</Value>
        </Stream>
        <Stream name="DontWalk" type="bool" class="output">
          <Value instant="0">true</Value>
          <Value instant="1">true</Value>
          <Value instant="2">false</Value>
          <Value instant="3">true</Value>
        </Stream>
        <Stream name="Phase" type="int" class="local">
          <Value instant="0">1</Value>
          <Value instant="1">2</Value>
          <Value instant="2">3</Value>
          <Value instant="3">1</Value>
        </Stream>
      </Node>
    </Node>
  </Counterexample>
</Property>

<Property name="R9">
  <Runtime unit="sec" timeout="false">0.162</Runtime>
  <K>2</K>
  <Answer source="ic3">falsifiable</Answer>
  <Counterexample>
    <Node name="ReqTrafficLight" >
      <Stream name="Button" type="bool" class="input">
        <Value instant="0">true</Value>
        <Value instant="1">true</Value>
      </Stream>
      <Stream name="R9" type="bool" class="output">
        <Value instant="0">true</Value>
        <Value instant="1">false</Value>
      </Stream>
      <Stream name="Yellow" type="bool" class="local">
        <Value instant="0">true</Value>
        <Value instant="1">true</Value>
      </Stream>
      <Node name="TrafficLight" line="43" column="41">
        <Stream name="Button" type="bool" class="input">
          <Value instant="0">true</Value>
          <Value instant="1">true</Value>
        </Stream>
        <Stream name="Red" type="bool" class="output">
          <Value instant="0">false</Value>
          <Value instant="1">false</Value>
```

```
        </Stream>
        <Stream name="Yellow" type="bool" class="output">
          <Value instant="0">true</Value>
          <Value instant="1">true</Value>
        </Stream>
        <Stream name="Green" type="bool" class="output">
          <Value instant="0">false</Value>
          <Value instant="1">false</Value>
        </Stream>
        <Stream name="Walk" type="bool" class="output">
          <Value instant="0">false</Value>
          <Value instant="1">false</Value>
        </Stream>
        <Stream name="DontWalk" type="bool" class="output">
          <Value instant="0">true</Value>
          <Value instant="1">true</Value>
        </Stream>
        <Stream name="Phase" type="int" class="local">
          <Value instant="0">1</Value>
          <Value instant="1">1</Value>
        </Stream>
      </Node>
    </Node>
  </Counterexample>
</Property>

<Property name="R10">
  <Runtime unit="sec" timeout="false">0.178</Runtime>
  <K>3</K>
  <Answer source="bmc">falsifiable</Answer>
  <Counterexample>
    <Node name="ReqTrafficLight" >
      <Stream name="Button" type="bool" class="input">
        <Value instant="0">true</Value>
        <Value instant="1">false</Value>
        <Value instant="2">true</Value>
      </Stream>
      <Stream name="R10" type="bool" class="output">
        <Value instant="0">true</Value>
        <Value instant="1">true</Value>
        <Value instant="2">false</Value>
      </Stream>
      <Stream name="Red" type="bool" class="local">
        <Value instant="0">false</Value>
        <Value instant="1">true</Value>
        <Value instant="2">false</Value>
```

```
      </Stream>
      <Stream name="Green" type="bool" class="local">
        <Value instant="0">false</Value>
        <Value instant="1">false</Value>
        <Value instant="2">false</Value>
      </Stream>
      <Node name="TrafficLight" line="43" column="41">
        <Stream name="Button" type="bool" class="input">
          <Value instant="0">true</Value>
          <Value instant="1">false</Value>
          <Value instant="2">true</Value>
        </Stream>
        <Stream name="Red" type="bool" class="output">
          <Value instant="0">false</Value>
          <Value instant="1">true</Value>
          <Value instant="2">false</Value>
        </Stream>
        <Stream name="Yellow" type="bool" class="output">
          <Value instant="0">true</Value>
          <Value instant="1">false</Value>
          <Value instant="2">true</Value>
        </Stream>
        <Stream name="Green" type="bool" class="output">
          <Value instant="0">false</Value>
          <Value instant="1">false</Value>
          <Value instant="2">false</Value>
        </Stream>
        <Stream name="Walk" type="bool" class="output">
          <Value instant="0">false</Value>
          <Value instant="1">false</Value>
          <Value instant="2">false</Value>
        </Stream>
        <Stream name="DontWalk" type="bool" class="output">
          <Value instant="0">true</Value>
          <Value instant="1">true</Value>
          <Value instant="2">true</Value>
        </Stream>
        <Stream name="Phase" type="int" class="local">
          <Value instant="0">1</Value>
          <Value instant="1">2</Value>
          <Value instant="2">1</Value>
        </Stream>
      </Node>
    </Node>
  </Counterexample>
</Property>
```

```
</Results>
```

7.2 Certificates

References

[1] F.-X. Dormoy. Scade 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*, pages 1–9, 2008.

[2] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *Advances in Computing Science—ASIAN'99*, pages 1–12. Springer, 1999.

[3] P. Raymond. Lustre v4 manual, 2000.

[4] F. P. Y. Régis-Gianas. Menhir reference manual, 2014.