Kind 2 User Documentation

Version

January 3, 2017

Contents

Kind 2

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.

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

Requirements

- Linux or Mac OS X,

- OCaml 4.03 or later,
- Menhir parser generator, and
- a supported SMT solver
  - CVC4,
  - Yices 2, or
  - Yices 1
  - Z3 (presently recommended),

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 IC3" test
```

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.

Docker

Kind 2 is available on docker.

Retrieving / updating the image

Install docker and then run

```
docker pull kind2/kind2:dev
```

Docker will retrieve the *layers* corresponding to the latest version of the Kind 2 repository, `develop` version. If you are interested in the latest release, run

```
docker pull kind2/kind2
```

instead.

If you want to update your Kind 2 image to latest one, simply re-run the `docker pull` command.

Running Kind 2 through docker

To run Kind 2 on a file on your system, it is recommended to mount the folder in which this file is as a volume. In practice, run

```
docker run -v <absolute_path_to_folder>:/lus kind2/kind2:dev <options> /lus/<your_file>
```

where

- `<absolute_path_to_folder>` is the absolute path to the folder your file is in,
- `<your_file>` is the lustre file you want to run Kind 2 on, and
- `<options>` are some Kind 2 options of your choice.

N.B.

- the fact that the path to your folder must be absolute is a docker constraint;
- mount point `/lus` is arbitrary and does not matter as long as it is consistent with the last argument `/lus/<your_file>`. To avoid name clashes with folders already present in the container however, it is recommended to use `/lus`;
- replace `kind2:dev` by `kind2` if you want to run the latest release of Kind2 instead of the `develop` version;
- `docker run` does **not** update your local Kind 2 image to the latest one: the appropriate `docker pull` command does.

Packaging your local version of Kind 2

At the top level of the Kind 2 repository is a `Dockerfile` you can use to build your own Kind 2 image. To do so, just run

```
docker build -t kind2-local .
```

at the root of the repository. `kind2-local` is given here as an example, feel free to call it whatever you want.

Note that building your own local Kind 2 image **does require access to the Internet**. This is because of the packages the build process needs to retrieve, as well as for downloading the z3 and cvc4 solvers.

Techniques

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

- k-induction
- invariant generation
- IC3

Compositional reasoning

When verifying a node `n`, *compositional reasoning* consists in abstracting the complexity of the subnodes of `n` by their contracts. The idea is that the contract has typically a lot less state than the node it specifies, which in addition to its own state contains that of its subnodes recursively.

Compositional reasoning thus improves the scalability of Kind 2 by taking advantage of information provided by the user to abstract the complexity away. When in compositional mode (`--composition true`), Kind 2 will abstract all calls (to subnodes that have a contract) in the top node and verify the resulting, abstract system.

A successful compositional proof of a node does not guarantee the correctness of the concrete (un-abstracted) node though, since the subnodes have not been verified. For this reason compositional reasoning is usually applied in conjunction with *modular reasoning*, discussed in the next section.

Modular reasoning

Modular reasoning is activated with the option `--modular true`. In this mode, Kind 2 will perform whatever type of analysis is specified by the other flags on **every node** of the hierarchy, bottom-up.

A timeout for *each analysis* can be specified using the `--timeout_analysis` flag. It can be used in conjunction with the *global timeout* given with the `--timeout` or `--timeout_wall` time.

Internally Kind 2 builds on previous analyses when starting a new one. For instance, by using the invariants previously discovered in subnodes of the node under analysis.

Refinement in compositional and modular analyses

An interesting configuration is

```
kind2 --modular true --compositional true ...
```

If `top` calls `sub` and we analyze `top`, it means we have previously analyzed `sub`. We are running in compositional mode so the call to `sub` is originally abstracted by its contract. Say the analysis fails with a counterexample. The countexample might be spurious for the concrete version of `sub`: the failure would not happen if we used the concrete call to `sub` instead of the abstract one.

Say now that when we analyzed `sub`, we proved that it is correct. In this case Kind 2 will attempt to *refine* the call to `sub` in top. That is, undo the abstraction and use the implementation of `sub` in a new analysis.

Note that since `sub` is known to be correct, it is stronger than its contract. More precisely, it accepts fewer execution traces than its contract does. Hence anything proved with the abstraction of `sub` is still valid after refinement, and Kind 2 will use these results right away.

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.

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.

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

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.

Properties and top level node

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 <bool_expr> ;
```

where `<bool_expr>` 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> ...
```

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).

Contracts

A contract `(A,G,M)` for a node is a set of assumptions `A`, a set of guarantees `G`, and a set of modes `M`. The semantics of contracts is given in the Contract semantics section, here we focus on the input format for contracts. Contracts are specified either locally, using

the *inline syntax*, or externally in a *contract node*. Both the local and external syntax have a body composed of *items*, each of which define

- a ghost variable / constant,
- an assumption,
- a guarantee,
- a mode, or
- an import of a contract node.

They are presented in detail below, after the discussion on local and external syntaxes.

Inline syntax

A local contract is a special comment between the signature of the node

```
node <id> (...) returns (...) ;
```

and its body. That is, between the `;` of the node signature and the `let` opening its body.

A local contract is a special block comment of the form

```
(*@contract
  [item]+
*)
```

or

```
/*@contract
  [item]+
*/
```

External syntax

A contract node is very similar to a traditional lustre node. The two differences are that

- it starts with `contract` instead of `node`, and
- its body can only mention *contract items*.

A contract node thus has form

```
contract <id> (<in_params>) returns (<out_params>) ;
let
  [item]+
tel
```

To use a contract node one needs to import it through an inline contract. See the next section for more details.

Contract items and restrictions

Ghost variables and constants A ghost variable (constant) is a stream that is local to the contract. That is, it is not accessible from the body of the node specified. Ghost variables (constants) are defined with the `var` (`const`) keyword. Kind 2 performs type inference for constants so in most cases type annotations are not necessary.

The general syntax is

```
const <id> [: <type>] = <expr> ;
var   <id>  : <type>  = <expr> ;
```

For instance:

```
const max = 42 ;
var ghost_stream: real = if input > max then max else input ;
```

Assumptions An assumption over a node `n` is a constraint one must respect in order to use `n` legally. It cannot mention the outputs of `n` in the current state, but referring to outputs under a `pre` is fine.

The idea is that it does not make sense to ask the caller to respect some constraints over the outputs of `n`, as the caller has no control over them other than the inputs it feeds `n` with. The assumption may however depend on previous values of the outputs produced by `n`.

Assumptions are given with the `assume` keyword, followed by any legal Boolean expression:

```
assume <expr> ;
```

Guarantees Unlike assumptions, guarantees do not have any restrictions on the streams they can mention. They typically mention the outputs in the current state since they express the behavior of the node they specified under the assumptions of this node.

Guarantees are given with the `guarantee` keyword, followed by any legal Boolean expression:

```
guarantee <expr> ;
```

Modes A mode `(R,E)` is a set of *requires* `R` and a set of *ensures* `E`. Requires have the same restrictions as assumptions: they cannot mention outputs of the node they specify in the current state. Ensures, like guarantees, have no restriction.

Modes are named to ease traceability and improve feedback. The general syntax is

```
mode <id> (
  [require <expr> ;]*
  [ensure  <expr> ;]*
) ;
```

For instance:

```
mode engaging (
  require true -> not pre engage_input ;
  require engage_input ;
  -- No ensure, same as `ensure true ;`.
) ;
mode engaged (
  require engage_input ;
  require false -> pre engage_input ;
  ensure  output <= upper_bound ;
  ensure  lower_bound <= output ;
) ;
```

Imports A contract import *merges* the current contract with the one imported. That is, if the current contract is `(A,G,M)` and we import `(A\',G\',M\')`, the resulting contract is `(A U A\', G U G\', M U M\')` where `U` is set union.

When importing a contract, it is necessary to specify how the instantiation of the contract is performed. This defines a mapping from the input (output) formal parameters to the actual ones of the import.

When importing contract `c` in the contract of node `n`, it is **illegal** to mention an output of `n` in the actual input parameters of the import of `c`. The reason is that the distinction between inputs and outputs lets Kind 2 check that the assumptions and mode requirements make sense, *i.e.* do not mention outputs of `n` in the current state.

The general syntax is

```
import <id> ( <expr>,* ) returns ( <expr>,* ) ;
```

For instance:

```
contract spec (engage, disengage: bool) returns (engaged: bool) ;
let ... tel

node my_node (
  -- Flags are "signals" here, but `bool`s in the contract.
```

```
  engage, disengage: real
) returns (
  engaged: real
) ;
(*@contract
  var bool_eng: bool = engage <> 0.0 ;
  var bool_dis: bool = disengage <> 0.0 ;
  var bool_enged: bool = engaged <> 0.0 ;

  var never_triggered: bool = (
    not bool_eng -> not bool_eng and pre never_triggered
  ) ;

  assume not (bool_eng and bool_dis) ;
  guarantee true -> (
    (not engage and not pre bool_eng) => not engaged
  ) ;

  mode init (
    require never_triggered ;
    ensure not bool_enged ;
  ) ;

  import spec (bool_eng, bool_dis) returns (bool_enged) ;
*)
let ... tel
```

Mode references Once a mode has been defined it is possible to *refer* to it with

```
::<scope>::<mode_id>
```

where `<mode_id>` is the name of the mode, and `<scope>` is the path to the mode in terms of contract imports.

In the example from the previous section for instance, say contract `spec` has a mode `m`. The inline contract of `my_node` can refer to it by

```
::spec::m
```

To refer to the `init` mode:

```
::init
```

A mode reference is syntactic sugar for the `requires` of the mode in question. So if mode `m` is

```
mode m (
  require <r_1> ;
  require <r_2> ;
  ...
  require <r_n> ; -- Last require.
  ...
) ;
```

then `::<path>::m` is exactly the same as

```
(<r_1> and <r_1> and ... and <r_n>)
```

N.B.: a mode reference * is a Lustre expression of type `bool` just like any other Boolean expression. It can appear under a `pre`, be used in a node call or a contract import, *etc.* * is only legal **after** the mode item itself. That is, no forward/self-references are allowed.

An interesting use-case for mode references is that of checking properties over the specification itself. One may want to do so to make sure the specification behaves as intended. For instance

```
mode m1 (...) ;
mode m2 (...) ;
mode m3 (...) ;

guarantee true -> ( -- `m3` cannot succeed to `m1`.
  (pre ::m1) => not ::m3
) ;
guarantee true -> ( -- `m1`, `m2` and `m3` are exclusive.
  not (::m1 and ::m2 and ::m3)
) ;
```

Merge, When, Activate and Restart

> **Disclaimer**: the first few examples of this section illustrating (unsafe) uses of `when` and `activate` are **not legal** in Kind 2. They aim at introducing the semantics of lustre clocks. As discussed below, they are only legal when used inside a `merge`, hence making them safe clock-wise.
>
> Also, `activate` and `restart` are actually not a legal Lustre v6 operator. They are however legal in Scade 6.

A `merge` is an operator combining several streams defined on **complementary** clocks. There is two ways to define a stream on a clock. First, by wrapping its definition inside a `when`.

```
node example (in: int) returns (out: int) ;
var in_pos: bool ; x: int ;
let
  ...
  in_pos = x >= 0 ;
  x = in when in_pos ;
  ...
tel
```

Here, `x` is only defined when `in_pos`, its clock, is `true`. That is, with `nil` the undefined value, a trace of execution of `example` sliced to `x` could be

step	in	in_pos	x
0	3	true	3
1	-2	false	nil
0	-1	false	nil
1	7	true	7
0	42	true	42

The second way to define a stream on a clock is to wrap a node call with the `activate` keyword. The syntax for this is

```
(activate <node_name> every <clock>)(<input_1>, <input_2>, ...)
```

For example, consider the following node:

```
node sum_ge_10 (in: int) returns (out: bool) ;
var sum: int ;
let
  sum = in + (0 -> pre sum) ;
  out = sum >= 10 ;
tel
```

Say now we call this node as follows:

```
node example (in: int) returns (...) ;
var tmp, in_pos: bool ;
```

```
let
  ...
  in_pos = in >= 0 ;
  tmp = (activate sum_ge_10 every in_pos)(in) ;
  ...
tel
```

That is, we want `sum_ge_10(in)` to tick iff `in` is positive. Here is an example trace of `example` sliced to `tmp`; notice how the internal state of `sub` (*i.e.* `pre sub.sum`) is maintained so that it does refer to the value of `sub.sum` *at the last clock tick of the* `activate`:

step	in	in_pos	tmp	sub.in	pre sub.sum	sub.sum
0	3	true	false	3	nil	3
1	2	true	false	2	3	5
2	-1	false	nil	nil	5	nil
3	2	true	false	2	5	7
4	-7	false	nil	nil	7	nil
5	35	true	true	35	7	42
6	-2	false	nil	nil	42	nil

Now, as mentioned above the `merge` operator combines two streams defined on **complimentary** clocks. The syntax of `merge` is:

```
merge( <clock> ; <e_1> ; <e_2> )
```

where `e_1` and `e_2` are streams defined on `<clock>` and `not <clock>` respectively, or on `not <clock>` and `<clock>` respectively.

Building on the previous example, say add two new streams `pre_tmp` and `safe_tmp`:

```
node example (in: int) returns (...) ;
var tmp, in_pos, pre_tmp, safe_tmp: bool ;
let
  ...
  in_pos = in >= 0 ;
  tmp = (activate sum_ge_10 every in_pos)(in) ;
  pre_tmp = false -> pre safe_tmp  ;
  safe_tmp = merge( in_pos ; tmp ; pre_tmp when not in_pos ) ;
  ...
tel
```

That is, `safe_tmp` is the value of `tmp` whenever it is defined, otherwise it is the previous value of `safe_tmp` if any, and `false` otherwise. The execution trace given above becomes

step	in	in_pos	tmp	pre_tmp	safe_tmp
0	3	true	false	false	false
1	2	true	false	false	false
2	-1	false	nil	false	false
3	2	true	false	false	false
4	-7	false	nil	false	false
5	35	true	true	false	true
6	-2	false	nil	true	true

Just like with uninitialized `pre`s, if not careful one can easily end up manipulating undefined streams. Kind 2 forces good practice by allowing `when` and `activate ... every` expressions only inside a `merge`. All the examples of this section above this point are thus invalid from Kind 2's point of view.

Rewriting them as valid Kind 2 input is not difficult however. Here is a legal version of the last example:

```
node example (in: int) returns (...) ;
var in_pos, pre_tmp, safe_tmp: bool ;
let
  ...
  in_pos = in >= 0 ;
  pre_tmp = false -> pre safe_tmp  ;
  safe_tmp = merge(
    in_pos ;
    (activate sum_ge_10 every in_pos)(in) ;
    pre_tmp when not in_pos
  ) ;
  ...
tel
```

Kind 2 supports resetting the internal state of a node to its initial state by using the construct restart/every. Writing

```
(restart n every c)(x1, ..., xn)
```

makes a call to the node `n` with arguments `x1`, ..., `xn` and every time the Boolean stream `c` is true, the internal state of the node is reset to its initial value.

In the example below, the node `top` makes a call to `counter` (which is an integer counter *modulo* a constant `max`) which is reset every time the input stream `reset` is true.

```
node counter (const max: int) returns (t: int);
let
  t = 0 -> if pre t = max then 0 else pre t + 1;
tel

node top (reset: bool) returns (c: int);
let
  c = (restart counter every reset)(3);
tel
```

A trace of execution for the node top could be:

step	reset	c
0	false	0
1	false	1
2	false	2
3	false	3
4	true	0
5	false	1
6	false	2
7	true	0
8	true	0
9	false	1

> Remark: This construction can be encoded in traditional Lustre by having a Boolean input for the reset stream for each node. However providing a built-in way to do it facilitates the modeling of complex control systems.

Restart and activate can also be combined in the following way:

```
(activate (restart n every r) every c)(a1, ..., an)
(activate n every c restart every r)(a1, ..., an)
```

These two calls are the same (the second one is just syntactic sugar). The (instance of the) node `n` is restarted whenever `r` is true and the *resulting call* is activated when the clock `c` is true. Notice that the restart clock `r` is also sampled by `c` in this call.

Enumerated data types in Lustre

```
type t = enum { A, B, C };
node n (x : enum { C1, C2 }, ...) ...
```

Enumerated datatypes are encoded as subranges so that solvers handle arithmetic constraints only. This also allows to use the already present quantifier instantiation techniques in Kind 2.

N-way merge

As in Lustre V6, merges can also be performed on a clock of a user defined enumerated datatype.

```
merge c
 (A -> x when A(c))
 (B -> w + 1 when B(c));
```

Arguments of merge have to be sampled with the correct clock. Clock expressions for merge can be just a clock identifier or its negation or `A(c)` which is a stream that is true whenever `c = A`.

Merging on a Boolean clock can be done with two equivalent syntaxes:

```
merge(c; a when c; b when not c);
```

```
merge c
  (true -> a when c)
  (false -> b when not c);
```

Partially defined nodes

Kind 2 allows nodes to define their outputs only partially. For instance, the node

```
node count (trigger: bool) returns (count: int ; error: bool) ;
(*@contract
  var once: bool = trigger or (false -> pre once) ;
  guarantee count >= 0 ;
  mode still_zero (
    require not once ;
    ensure count = 0 ;
  ) ;
  mode gt (
    require not ::still_zero ;
    ensure count > 0 ;
  ) ;
*)
```

```
let
  count = (if trigger then 1 else 0) + (0 -> pre count) ;
tel
```

can be analyzed: first for mode exhaustiveness, and the body is checked against its contract, although it is only *partially* defined. Here, both will succeed.

The `imported` keyword

Nodes (and functions, see below) can be declared `imported`. This means that the node does not have a body (`let ... tel`). In a Lustre compiler, this is usually used to encode a C function or more generally a call to an external library.

```
node imported no_body (inputs: ...) returns (outputs: ...) ;
```

In Kind 2, this means that the node is always abstract in the contract-sense. It can never be refined, and is always abstracted by its contract. If none is given, then the implicit (rather weak) contract

```
(*@contract
  assume true ;
  guarantee true ;
*)
```

is used.

In a modular analysis, `imported` nodes will not be analyzed, although if their contract has modes they will be checked for exhaustiveness, consistently with the usual Kind 2 contract workflow.

Partially defined nodes VS `imported`

Kind 2 allows partially defined nodes, that is nodes in which some streams do not have a definition. At first glance, it might seem like a node with no definitions at all (with an empty body) is the same as an `imported` node.

It is not the case. A partially defined node *still has a (potentially empty) body* which can be analyzed. The fact that it is not completely defined does not change this fact. If a partially defined node is at the top level, or is in the cone of influence of the top node in a modular analysis, then it's body **will** be analyzed.

An `imported` node on the other hand *explicitely does not have a body.* Its non-existent body will thus never be analyzed.

Functions

Kind 2 supports the `function` keyword which is used just like the `node` one but has slightly different semantics. Like the name suggests, the output(s) of a `function` should be a *non-temporal* combination of its inputs. That is, a function cannot the `->`, `pre`, `merge`, `when`, `condact`, or `activate` operators. A function is also not allowed to call a node, only other functions. In Lustre terms, functions are stateless.

In Kind 2, these retrictions extend to the contract attached to the function, if any. Note that besides the ones mentioned here, no additional restrictions are enforced on functions compared to nodes.

Benefits

Functions are interesting in the model-checking context of Kind 2 mainly as a mean to make an abstraction more precise. A realistic use-case is when one wants to abstract non-linear expressions. While the simple expression `x*y` seems harmless, at SMT-level it means bringing in the theory of non-linear arithmetic.

Non-linear arithmetic has a huge impact not only on the performances of the underlying SMT solvers, but also on the SMT-level features Kind 2 can use (not to mention undecidability). Typically, non-lineary arithmetic tends to prevent Kind 2 from performing satisfiability checks with assumptions, a feature it heavily relies on.

The bottom line is that as soon as some non-linear expression appear, Kind 2 will most likely fail to analyze most non-trivial systems because the underlying solver will simply give up.

Hence, it is usually extremely rewarding to abstract non-linear expressions away in a separate *function* equipped with a contract. The contract would be a linear abstraction of the non-linear expression that is precise enough to prove the system using correct. That way, a compositional analysis would *i)* verify the abstraction is correct and *ii)* analyze the rest of the system using this abstraction, thus making the analysis a linear one.

Using a function instead of a node simply results in a better abstraction. Kind 2 will encode, at SMT-level, that the outputs of this component depend on the *current* version of its inputs only, not on its previous values.

Hierarchical Automata

Experimental feature

Kind 2 supports both the syntax used in LustreC and a subset of the one used in Scade 6.

```
node n (i1, ..., in : ...) returns (o1, ..., on : ...);
```

```
let

   automaton automaton_name

     initial state S1:
       unless if c restart Si elsif c\' resume Sj else restart Sk end;
       var v : ...;
       let
          v = ...;
          o1 = i1 -> last o2 + 1;
          o2 = 99;
       tel
       until c restart S2;

     state S2:
       let
          ...;
       tel
     ...
   returns o1, o2;

   o3 = something () ...;
tel
```

An automaton is declared *inside a node* (there can be several) and can be anonymous. Automata can be nested, *i.e.* an automaton can contain other automata in some of its states bodies. This effectively allows to describe *hierarchical state machines*. An automaton is defined by its list of states and a `returns` statement that specifies which variables (locals or output) are defined by the automaton.

> The set of returned streams can be inferred by writing `returns ..;`. One can also simply omit the `returns` statement which will have the same effect.

States (much like regular nodes) do not need to give equations that define *all* their outputs (but they do for their local variables). If defined streams are different between the states of the automaton, then the set considered will be their union and states that do not define all the inferred streams will be considered underconstrained.

Each state has a name and one of them can be declared `initial` (if no initial state is specified, the first one is considered initial). They can have local variables (to the state). The body of the state contains Lustre equations (or assertions) and can use the operator `last`. In contrast to `pre x` which is the value of the stream `x` the last time the state was in the declared state, `last x` (or the Scade 6 syntax `last \'x`) is the previous value of the stream `x` on the base clock. This construct is useful for communicating information between states.

States can have a *strong* transition (declared with `unless`) placed before the body and a *weak* transition placed after the body. The unless transition is taken when entering the state, whereas the until transition is evaluated after execution of the body of the state. If none are applicable then the automaton remains in the same state. These transitions express conditions to change states following a branching pattern. Following are examples of legal branching patterns (`c*` are Lustre Boolean expressions):

```
c restart S
```

```
if c1 restart S1
elsif c2 restart S2
elsif c3 restart S3
end;
```

```
if c1
  if c2 restart S2
  else if c3 resume S1
  end
elsif c3 resume S3
else restart S0
end;
```

Targets are of the form `restart State_name` or `resume State_name`. When transiting to a state with `restart`, the internal state of the state is rested to its initial value. On the contrary when transiting with `resume`, execution in the state resumes to where it was when the state was last executed.

In counter-examples, we show the value of additional internal state information for each automaton: `state` is a stream that denotes the state in which the automaton is and `restart` indicates if the state in which the automaton is was restarted in the current instant.

The internal state of an automaton state is also represented in counter-example traces, separately. States and subsequent streams are sampled with the clock state, *i.e.* values of streams are shown only when the automaton is in the corresponding state.

Arrays

Experimental feature

Lustre arrays

Kind 2 supports the traditional Lustre V5 syntax for arrays.

Declarations

Array variables can be declared as global, local or as input/output of nodes. Arrays in Lustre are always indexed by integers (type `int` in Lustre), and the type of an array variable is written with the syntax `t ^ <size>` where `t` is a Lustre type and `<size>` is an integer literal or a constant symbol.

The following

```
A : int ^ 3;
```

declares an array variable `A` of type array of size 3 whose elements are integers. The size of the array can also be given by a defined constant.

```
const n = 3;
...
A : int ^ n;
```

This declaration is equivalent to the previous one for `A`.

An interesting feature of these arrays is the possibility for users to write generic nodes and functions that are parametric in the size of the array. For instance one can write the following node returns the last element of an array.

```
node last (const n: int; A: int ^ n) returns (x: int);
let
  x = A[n-1];
tel
```

It takes as input the size of the array and the array itself. Note that the type of the input `A` depends on the value of the first constant input `n`. In Lustre, calls to such nodes should of course end up by having concrete values for `n`, this is however not the case in Kind 2 (see here).

Arrays can be multidimensional, so a user can declare *e.g.* matrices with the following

```
const n = 4;
const m = 5;
...

M1 : bool ^ n ^ m;
M2 : int ^ 3 ^ 3;
```

Here `M1` is a matrix of size 4x5 whose elements are Boolean, and `M2` is a square matrix of size 3x3 whose elements are integers.

Remark

`M1` can also be viewed as an array of arrays of Booleans.

Kind 2 also allows one to nest datatypes, so it is possible to write arrays of records, records of arrays, arrays of tuples, and so on.

```
type rational = { n: int; d: int };

rats: rational^array_size;
mm: [int, bool]^array_size;
```

In this example, `rats` is declared as an array of record elements and `mm` is an array of pairs.

Definitions

In the body of nodes or at the top-level, arrays can be defined with literals of the form

```
A = [2, 5, 7];
```

This defines an array `A` of size 3 whose elements are 2, 5 and 7. Another way to construct Lustre arrays is to have each elements be the same value. This can be done with expressions of the form `<value> ^ <size>`. For example the two following definitions are equivalent.

```
A = 2 ^ 3;
A = [2, 2, 2];
```

Arrays are indexed starting at 0 and the elements can be accessed using the selection operator `[ ]`. For instance the result of the evaluation of the expression `A[0]` for the previously defined array `A` is 2.

The selection operators can also be applied to multidimensional arrays. Given a matrix `M` defined by

```
M = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]];
```

then the expression `M[1][2]` is valid and evaluates to 6. The result of a single selection on an *n*-dimensional array is an *(n-1)*-dimensional array. The result of `M[2]` is the array `[7, 8, 9]`.

Unsupported features of Lustre V5

Kind 2 currently **does not support** the following features of Lustre V5:

- Array concatenation like `[0, 1] | [2, 3, 4]`
- Array slices like `A[0..3]`, `A[0..3 step 2]`, `M[0..1][1..2]` or `M[0..1, 1..2]`
- The operators are not homomorphically extended. For instance `or` has type `bool -> bool -> bool`, given two arrays of Booleans `A` and `B`, the expression `A or B` will be rejected at typing by Kind 2
- Node calls don't have an homomorphic extension either

Extension to unbounded arrays

Kind 2 provides an extension of Lustre to express equational constraints between unbounded arrays. This syntax extension allows users to inductively define arrays, give whole array definitions and allows to encode most of the other unsupported array features. This extension was originally suggested by Esterel.

> **Remark**
>
> Here, by *unbounded* we mean whose size is an unbounded constant.

In addition, we also enriched the specification language of Kind 2 to support (universal and existential) quantifiers, allowing one to effectively model *parameterized* system.

Whole array definitions

Equations in the body of nodes can now take the following forms

- `A[2] = <term> ;` This equation defines the element at index `2` to the value `<term>`, the other elements are undefined. This form of *single point* updates can only be written when the selections on the left of the equation are at constant literal integers.

- `A[i] = <term(i)> ;` This equation defines the values of all elements in the array `A`. The index `i` has to be a symbol, it is bound locally to the equation and shadows all other mentions of `i`. Index variables that appear on the left hand side of equations are **implicitly universally quantified**. The right hand side of the equation, `<term(i)>` can depend on this index. The meaning of the equation is that, for any integer `i` between 0 and the size of `A`, the value at position `i` is defined as the term `<term(i)>`.
- `A = B ;` This equation defines the values of the array `A` to be the same as the values of `B`. This equation is equivalent to the equation `A[i] = B[i] ;`.

Semantically, a whole array equation is equivalent to a quantified equation. Let `A` be an array of size an integer constant `n`, then following equation is legal.

```
A[i] = if i = 0 then 2 else B[i - 1] ;
```

It is equivalent to the formula $\forall\ i \in [0; n].\ (\ i = 0 \Rightarrow A[i] = 2\ ) \bigwedge (\ i \neq 0 \Rightarrow A[i] = B[i\text{-}1]\ )$.

Multidimensional arrays can also be redefined the same way. For instance the equation

```
M[i][j] = if i = j then 1 else 0 ;
```

defines `M` as the identity matrix

```
[[ 1 , 0 , 0 ,..., 0 ],
 [ 0 , 1 , 0 ,..., 0 ],
 [ 0 , 0 , 1 ,..., 0 ],
 ..................... ,
 [ 1 , 0 , 0 ,..., 1 ]]
```

It is possible to write an equation of the form

```
M[i][i] = i;
```

but in this case the second index `i` shadows the first one, hence the definition is equivalent to the following one where the indexes have been renamed.

```
M[j][i] = i;
```

Inductive definitions

One interesting feature of these equations is that we allow definitions of arrays *inductively*. For instance it is possible to write an equation

```
A[i] = if i = 0 then 0 else A[i-1] ;
```

This is however not very exciting because this is the same as saying that `A` will contain only zeros, but notice we allow the use of `A` in the right hand side.

Dependency analysis Inductive definitions are allowed under the restriction that they should be *well founded*. For instance, the equation

```
A[i] = A[i];
```

is not and will be rejected by Kind 2 the same way the equation `x = x;` is rejected. Of course this restriction does not apply for array variables under a `pre`, so the equation `A[i] = pre A[i];` is allowed.

In practice, Kind 2 will try to prove statically that the definitions are well-founded to ensure the absence of dependency cycles. We only attempt to prove that definitions for an array `A` at a given index `i` depends on on values of `A` at indexes strictly smaller than `i`.

For instance the following set of definitions is rejected because *e.g.* `A[k]` depends on `A[k]`.

```
A[k] = B[k+1] + y;
B[k] = C[k-1] - 2;
C[k] = A[k] + k;
```

On the other hand this one will be accepted.

```
A[k] = B[k+1] + y;
B[k] = C[k-1] - 2;
C[k] = ( A[k-1] + B[k] ) * k ;
```

Because the order is fixed and that the checks are simple, it is possible that Kind 2 rejects programs that are well defined (with respect to our semantic for whole array updates). It will not, however, accept programs that are ill-defined.

For instance each of the following equations will be rejected.

```
A[i] = if i = 0 then 0 else if i = 1 then A[0] else A[i-1];

A[i] = if i = n then 0 else A[i+1];

A[i] = if i = 0 then 0 else A[0];
```

Examples This section gives some examples of usage for inductive definitions and whole array updates as a way to encode unsupported features and as way to encode complicated functions succinctly.

Sum of the elements in an array The following node returns the sum of all elements in an array.

```
node sum (const n: int; A: int ^ n) returns (s: int);
var cumul: int ^ n;
let
  cumul[i] = if i = 0 then A[0] else A[i] + cumul[i-1];
  s = cumul[n-1];
tel
```

We declare a local array `cumul` to store the cumulative sum (*i.e.* `cumul[i]` contains the sum of elements in `A` up to index `i`) and the returned value of the node is the element stored in the last position of `cumul`.

Note that this node is parametric in the size of the array.

Array slices Array slices can be trivially implemented with the features presented above.

```
node slice (const n: int; A: int ^ n; const low: int; const up: int)
returns (B : int ^ (up-low));
let
  B[i] = A[low + i];
tel
```

Homomorphic extensions Encoding an homomorphic `or` on Boolean arrays is even simpler.

```
node or_array (const n: int; A, B : bool^n) returns (C: bool^n);
let
  C[i] = A[i] or B[i];
tel
```

Defining a generic homomorphic extension of node calls is not possible because nodes are not first order objects in Lustre.

Parameterized systems It is possible to describe and check properties of parameterized systems. Contrary to the Lustre compilers, Kind 2 does not require the constants used as array sizes to be instantiated with actual values. In this case the properties are checked *for any* array sizes.

```
node slide (const n:int; s: int) returns(A: int^n);
let
  A[i] = if i = 0 then s else (-1 -> pre A[i-1]);

  --%PROPERTY n > 1 => (true -> A[1] = pre s);
tel
```

This node stores in an array `A` a *sliding window* over an integer stream `s`. It saves the values taken by `s` up to `n` steps in the past, where `n` is the size of the array.

Here the property says, that if the array `A` has at least two cells then its second value is the previous value of `s`.

Quantifiers in specifications

To better support parameterized systems or systems with large arrays, we expose quantifiers for use in the language of the specifications. Quantifiers can thus appear in **properties**, **contracts** and **assertions**.

Universal quantification is written with:

```
forall ( <x : type>;+ ) P(<x>+)
```

where `x` are the quantified variables and `type` is their type. `P` is a formula or a predicate in which the variable `x` can appear.

For example, the following

```
forall (i, j: int) 0 <= i and i < n and 0 <= j and j < n => M[i][j] = M[j][i]
```

is a formula that specifies that the matrix `M` is symmetric.

> **Remark**
>
> Existential quantification takes the same form except we use `exists` instead of `forall`.

Quantifiers can be arbitrarily nested and alternated at the propositional level.

Example The same parameterized system of a sliding window, slightly modified to express the property that `A` contains in each of its cells, an uninitialized value (*i.e.* value `-1`), or one of the previous values of the stream `s`.

```
node slide (const n:int; s: int) returns(ok: bool^n);
var A: int^n;
let
  A[i] = if i = 0 then s else (-1 -> pre A[i-1]);
  ok[i] = A[i] = -1 or A[i] = s or (false -> pre ok[i]);

  --%PROPERTY forall (i: int) 0 <= i and i < n => ok[i];
tel
```

Limitations

One major limitation that is present in the arrays of Kind 2 is that one cannot have node calls in inductive array definitions whose parameters are array selections.

For instance, it is currently not possible to write the following in Kind 2 where `A` and `B` are array and `some_node` takes values as inputs.

```
node some_node (x: int) returns (y: int);
...

A, B: int^4;
...

A[i] = some_node(B[i]);
```

This limitation exists only for technical implementation reasons. A workaround for the moment is to redefine an homorphic extension of the node and use that instead.

```
node some_node (const n: int; x: int^n) returns (y: int^n);
...

A, B: int^4;
...

A = some_node(4, B);
```

Command line options

We provide different encodings of inductive array definitions in our internal representation of the transition system. The command line interface exposes different options to control which encoding is used. This is particularly relevant for SMT solvers that have built-in features, whether it is support for the theory of arrays, or special options or annotations for quantifier instantiation.

These options are summed up in the following table and described in more detail in the rest of this section.

Option	Description
`--smt_arrays`	Use the builtin theory of arrays in solvers
`--inline_arrays`	Instantiate quantifiers over array bounds in case they are statically known
`--arrays_rec`	Define recurvsive functions for arrays (for CVC4)

The default encoding will use quantified formulas for inductive definitions and whole array updates.

For example if we have

```
A : int^6;
...
A[k] = x;
```

we will generate internally the constraint

$\forall$ *k: int. 0 <= k < 6 => (select A k) = x*

These form of constraint are handled in an efficient way by CVC4 (thanks to finite model finding).

`--smt_arrays` By default arrays are converted using ah-hoc selection functions to avoid stressing the theory of arrays in the SMT solvers. This option tells Kind 2 to use the builtin theory of arrays of the solvers instead. If you want to try it, it's probably a good idea to use it in combination of `--smtlogic detect` for better performances.

`--inline_arrays` By default, Kind 2 will generate problems with quantifiers for arrays which should be useful for problems with large arrays. This option tells Kind 2 to instantiate these quantifiers when it can reasonably do so. Only CVC4 has a good support for this kind of quantification so you may want to use this option with the other solvers.

The previous example

```
A : int^6;
...
A[k] = x;
```

will now be encoded by the constraint

(select A 0) = x $\bigwedge$ *(select A 1) = x* $\bigwedge$ *(select A 2) = x* $\bigwedge$ *(select A 3) = x* $\bigwedge$ *(select A 4) = x* $\bigwedge$ *(select A 5) = x*

`--arrays_rec` This uses a special kind of encoding to tell CVC4 to treat quantified definitions of some uninterpreted functions as recursive definitions.

XML Output

The XML output is activated by running Kind 2 with the `-xml` option. It is fully specified by file `xmlschema.xsd` located in folder `XMLSchema` on the repository.

Post Analyses Treatments

Post-analysis treatments are flag-activated Kind 2 features that are not directly related to verification. The current post-analysis treatments available are

- certification,
- compilation to Rust,
- test generation,
- contract generation, and
- invariant logging.

All of them are deactivated by default. Post-analysis treatments run on the *last analysis* of a system. It is defined as the last analysis performed by Kind 2 on a given system. With the default settings, Kind 2 performs a single, monolithic analysis of the top node. In this case, the *last analysis* is this unique analysis.

This behavior is changed by the `compositional` flag. For example, say Kind 2 is asked to analyze node `top` calling two subnodes `sub_1` and `sub_2`, in compositional mode. Say also `sub_1` and `sub_2` have contracts, and that refinement is possible. In this situation, Kind 2 will analyze `top` by abstracting its two subnodes. Assume for now that this analysis concludes the system is safe. Kind 2 has nothing left to do on `top`, so this compositional analysis is the *last analysis* of `top`, Kind 2 will run the post-analysis treatments. Assume now that this purely compositional analysis discovers a counterexample. Since refinement is possible, Kind 2 will refine `sub_1` (and/or `sub_2`) and start a new analysis. Hence, the first, purely compositional analysis is not the *last analysis* of `top`. The analysis where `sub_1` and `sub_2` are refined is the *last analysis* of `top` regardless of its outcome (assuming no other refinement is possible).

Long story short, the *last analysis* of a system is either the first analysis allowing to prove the system safe, or the analysis where all refineable systems have been refined.

The `modular` flag forces Kind 2 to apply whatever analysis / treatment the rest of the flags specify to all the nodes of the system, bottom-up. Post-analysis treatments respect this behavior and will run on the last analysis of each node.

Prerequisites

Some treatments can fail (which results in a warning) because some conditions were not met by the system and/or the last analysis. The prerequisites for each treatment are:

Treatment	Condition	Notes
certification	last analysis proved the system safe	
compilation to Rust	none	can fail if node is partially defined
test generation	system has a contract with more than one mode	

Treatment	Condition	Notes
	last analysis proved the system safe	
contract generation	none	
invariant logging	last analysis proved the system safe	

Silent Contract Loading

Two of the treatments mentioned above end up, if successful, generating a contract for the current node: invariant logging and contract generation. The natural way to benefit from these contracts is to import them explicitely in the original system.

If you do not import these contracts however, *silent contract loading* will still try to take advantage of them. That is, contracts logged by Kind 2 in previous runs will be loaded as *candidate properties*. A candidate property is similar to a normal property except that it is allowed to be falsifiable. That is, falsification of a candidate property does not impact the safety of the system under analysis.

Note that if you change the signature of the system, silent contract loading may fail. This failure is silent, and simply results in Kind 2 analyzing the system without any candidates.

> **NB**: for silent contract loading to work, it needs to be able to find the contracts. In practice, Kind 2 will look in the *output directory* specified by `--output_dir`, the default being `<input_file>.out/`.
>
> Kind 2 writes the contracts resulting from invariant logging and contract generation in the output directory, in a sub-directory named after the system the contract is for.
>
> As a result, running `kind2 --log_invs on --lus_main top example.lus` will log invariants in `example.lus.out/top/kind2_strengthening.lus`. Running the same command again will cause Kind 2 to silently load this contract as candidates.
>
> If one changes the output directory though, for instance by running `kind2 --output_dir out --log_invs on --lus_main top example.lus`, then silent contract loading will not find the contracts written to `example.lus.out/top` because it will look in `out/top`.
>
> Running `kind2 --output_dir out --log_invs on --lus_main top example.lus` two times however results in Kind 2 silently loading the contract generated during the first analysis in `out/top` on the second run.

Contract semantics

Assume-guarantee contracts

This section discusses the semantics of contracts, and in particular modes, in Kind 2. For details regarding the syntax, please see the contract syntax section.

An *assume-guarantee contract* `(A,G)` for a node `n` is a set of *assumptions* `A` and a set of *guarantees* `G`. Assumptions describe how `n` **must** be used, while guarantees specify how `n` behaves.

More formally, `n` respects its contract `(A,G)` if

```
([] A) => ([] G)
```

where `[]` is the box (globally) temporal operator.

That is, if the assumptions always hold then the guarantees hold. Contracts are interesting when a node `top` calls a node `sub`, where `sub` has a contract `(A,G)`.

From the point of view of `sub`, a contract `({a_1, ..., a_n}, {g_1, ..., g_m})` represents the same verification challenge as if `sub` had been written

```
node sub (...) returns (...) ;
let
  ...
  assert a_1 ;
  ...
  assert a_n ;
  --%PROPERTY g_1 ;
  ...
  --%PROPERTY g_m ;
tel
```

The guarantees must be invariant of `sub` when the assumptions are forced.

For the caller however, the call `sub(<params>)` is legal **if and only if** the assumptions of `sub` are invariants of `top` at call-site. The verification challenge for `top` is therefore the same as

```
node top (...) returns (...) ;
let
  ... sub(<params>) ...
  --%PROPERTY a_1(<call_site>) ;
  ...
  --%PROPERTY a_n(<call_site>) ;
tel
```

Modes

Kind 2 augments traditional assume-guarantee contracts with the notion of *mode*. A mode `(R,E)` is a set `R` or *requires* and a set `E` of *ensures*. A Kind 2 contract is therefore a triplet `(A,G,M)` where `M` is a set of modes. If `M` is empty then the semantics of the contract is exactly that of an assume-guarantee contract.

Semantics

A mode represents a *situation / reaction* implication. A contract `(A,G,M)` can be re-written as an assume-guarantee contract `(A,G\')` where

```
G\' = G U { (/\ r_i) => (/\ e_i), ({r_i}, {e_i}) in M }
```

where `U` is set union.

For instance, a (linear) contract for non-linear multiplication could be

```
node abs (in: real) returns (res: real) ;
let res = if in < 0.0 then - in else in ; tel

node times (lhs, rhs: real) returns (res: real) ;
(*@contract

  mode absorbing (
    require lhs = 0.0 or rhs = 0.0 ;
    ensure res = 0.0 ;
  ) ;
  mode lhs_neutral (
    require not absorbing ;
    require abs(lhs) = 1.0 ;
    ensure abs(res) = abs(rhs) ;
  ) ;
  mode rhs_neutral (
    require not absorbing ;
    require abs(rhs) = 1.0 ;
    ensure abs(res) = abs(lhs) ;
  ) ;
  mode positive (
    require (
      rhs > 0.0 and lhs > 0.0
    ) or (
      rhs < 0.0 and lhs < 0.0
```

```
    ) ;
    ensure res > 0.0 ;
  ) ;
  mode pos_neg (
    require (
      rhs > 0.0 and lhs < 0.0
    ) or (
      rhs < 0.0 and lhs > 0.0
    ) ;
    ensure res < 0.0 ;
  ) ;
*)
let
  res = lhs * rhs ;
tel
```

Motivation: modes were introduced in the contract language of Kind 2 to account for the fact that most requirements found in specification documents are actually implications between a situation and a behavior. In a traditional assume-guarantee contract, such requirements have to be written as `situation => behavior` guarantees. We find this cumbersome, error-prone, but most importantly we think some information is lost in this encoding. Modes make writing specification more straightforward and user-friendly, and allow Kind 2 to keep the mode information around to * improve feedback for counterexamples, * generate mode-based test-cases, and * adopt a defensive approach to guard against typos and specification oversights to a certain extent. This defensive approach is discussed in the next section.

Defensive check

Conceptually modes correspond to different situations triggering different behaviors for a node. Kind 2 is *defensive* in the sense that when a contract has at least one mode, it will check that the modes account for **all situations** the assumptions allow before trying to prove the node respects its contract.

More formally, consider a node `n` with contract

```
(A, G, M = { (r_i, e_i) }
```

The defensive check consists in checking that the disjunction of the requires of each mode

```
one_mode_active = \/ { r_i }
```

is an invariant for the system

```
A /\ G /\ { r_i => e_i }
```

If `one_mode_active` is indeed invariant, it means that as long as * the assumptions are respected, and * the node is correct *w.r.t.* its contract then at least one mode is active at all time.

Kind 2 follows this defensive approach. If a mode is missing, or a requirement is more restrictive than it should be then Kind 2 will detect the modes that are not exhaustive, provide a counterexample and stop.

This defensive approach is not as constraining as it first appears. If one wants to leave some situation unspecified on purpose, it is enough to add to the current set of (non-exhaustive) modes a mode like

```
mode base_case (
  require true ;
) ;
```

which explicitly accounts for, and hence documents, the missing cases.

Test generation

> Test generation is, as of Kind 2 1.0, still a rather experimental feature. There is a lot of room for improvement and the Kind 2 team is eagerly awaiting feedback / bug reports.

Most test generation techniques analyze the syntax of the model they run on to generate test cases satisfying some coverage criteria. Kind 2 does not follow this approach but instead generates tests based on the specification, more precisely the *modes* of the specification.

Kind 2's test generation was developed in a context where the actual implementation of the components is **outsourced**. That is, a *model* of the system is written in-house based on some specification. The model is then verified correct with respect to its specification, using Kind 2 of course, before the specification is given to external sub-contractors that will eventually produce some binaries but will **not** give access to their source code. At this point, there is a need to test these binaries in-house.

In this context, syntactic test generation is arguably not appropriate as it would be based on the syntax of the *model*, not that of the actual source code of the binaries. There is no reason to believe any connection between the two. Now, the only thing we know of the binaries is that they are supposed to verify the specification. For this reason, Kind 2's test generation ignores the syntax of the input model and instead builds on contracts, and more precisely on the notion on *mode*.

Combinations of modes as abstractions

Modes specify behaviors specific to a situation in a contract, and can be seen as abstractions of the states allowed by the assumptions of the contract. Note that because of the *mode exhaustiveness check*, there is always at least one mode active in any reachable state.

One can explore, starting from the initial states, the mode that can be activated up to some depth. For example, consider the following `stopwatch` system:

```
contract stopwatchSpec ( tgl, rst : bool ) returns ( c : int ) ;
let
  var on: bool = tgl -> (pre on and not tgl) or (not pre on and tgl) ;
  assume not (rst and tgl) ;
  guarantee c >= 0 ;
  mode resetting ( require rst ; ensure c = 0 ; ) ;
  mode running (
    require not rst ; require on ; ensure c = (1 -> pre c + 1) ;
  ) ;
  mode stopped (
```

```
    require not rst ; require not on ; ensure c = (0 -> pre c) ;
  ) ;
tel

node previous ( x : int ) returns ( y : int ) ;
let
  y = 0 -> pre x ;
tel

node stopwatch ( toggle, reset : bool ) returns ( count : int ) ;
(*@contract
  import stopwatchSpec ( toggle, reset ) returns ( count ) ;
*)
var running : bool ;
let
  running = (false -> pre running) <> toggle ;
  count = if reset then 0 else
    if running then previous(count) + 1 else previous(count) ;
tel
```

It seems that any of the three modes from the contract can be active at any point, since their activation only depends on the values of the inputs. We can ask Kind 2 to generate the graph of mode paths up to some depth (5 here):

```
kind2 --testgen on --testgen_len 5 stopwatch.lus
```

This will generate the following graph (and a lot of other files we will discuss below but omit for now):

The graph confirms our understanding of the specification, each mode can be activated at any time. Say now we made a mistake on the assumption:

```
  assume not (rst or tgl) ;
```

It is now illegal to reset or start the stopwatch. The graph is generated very quickly as with this assumption the system cannot do anything:

N.B. In this simple system, only one mode could be active at a time. This is not the case in general. See for example the mode graphs for the mode logic or the full model of the Transport Class Model (TCM) case study.

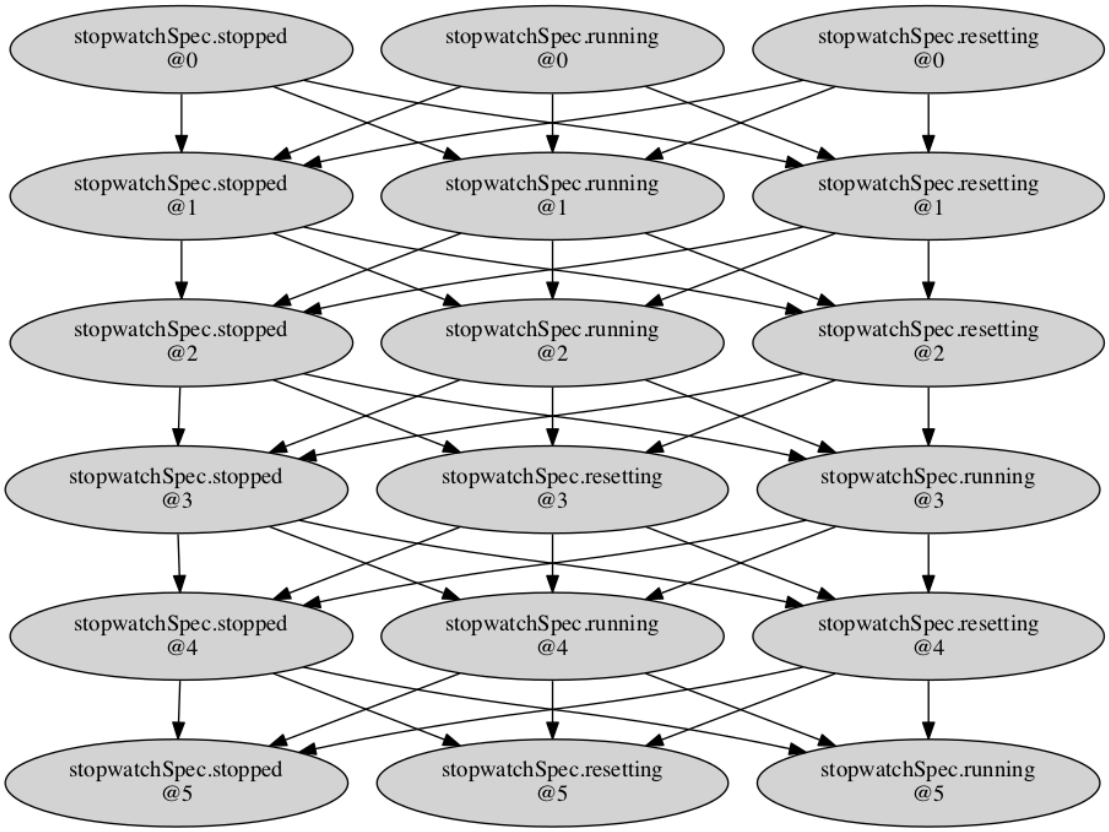

Figure 1: Stopwatch DAG

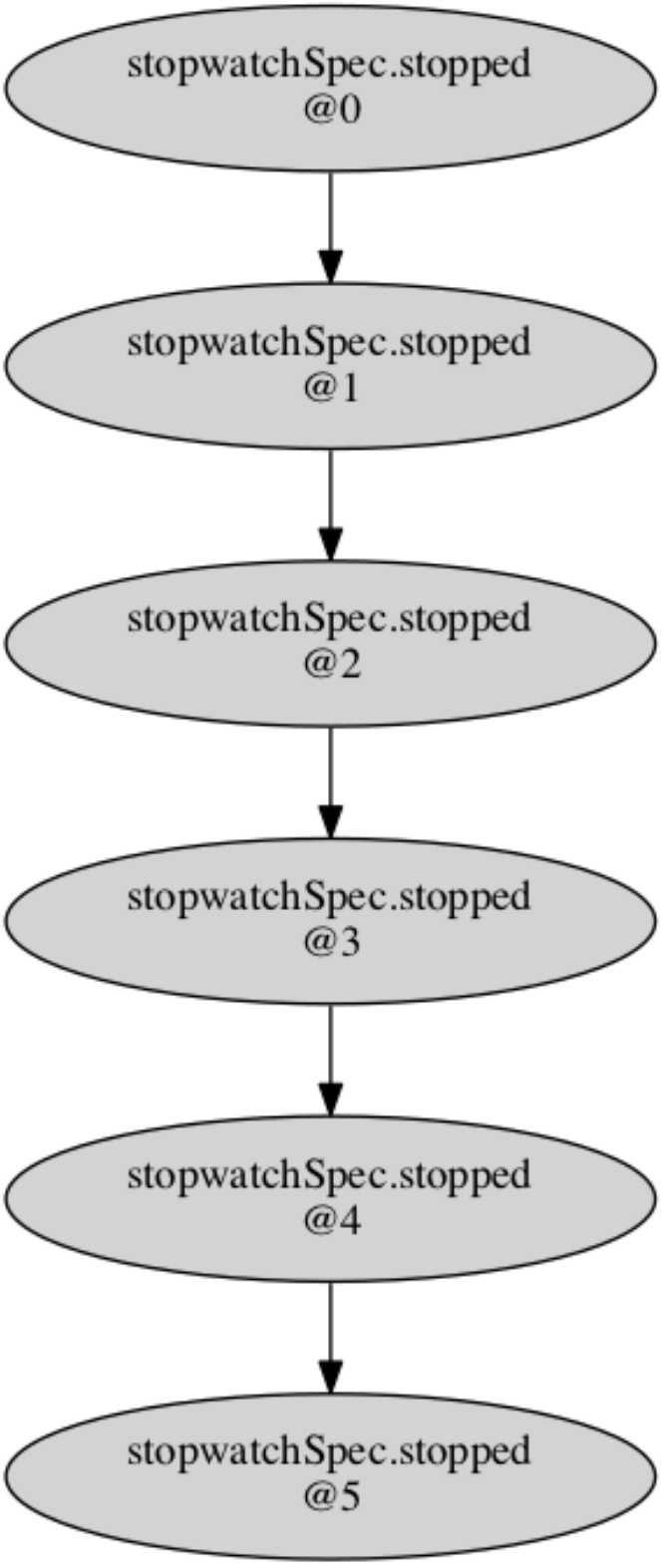

Figure 2: Stopwatch mistake DAG

Generating test cases

Since Kind 2 can explore the traces of combinations of modes that can be activated from the initial states, generating test cases is simple. Each test case is simply a trace of inputs, or *witness*, triggering a different path of mode combinations in the DAG discussed above.

Each witness is logged in CSV file. A glue XML file lists all the test cases and provides additional information such as the trace of mode combinations they triggered in the model.

> But aren't the witnesses still based on how the model is written?

Yes they are. There is no way to completely abstract the model/prototype away, nor is it desirable. Generating test cases solely on the specification is not realistic unless the specification is extremely strong and precise, which it very rarely is. (Also, if it was, it would arguably be easier to produce the object code as a refinement of the specification using B-method for instance.)

Oracle generation

The point of generating these test cases is to eventually run them on an executable version of the model to check whether it crashes and respects the specification.

For convenience, Kind 2 automatically generates an executable *oracle* along with the test cases. It takes the form of a Rust project in the `oracle` subdirectory of the Kind 2 output directory. The best way to learn about how this oracle behaves is to generate and read its documentation by running `cargo doc` in said subdirectory and opening `target/doc/<system>/index.html`.

The idea is that this oracle will read comma-separated values on its standard input. These values correspond to the inputs fed to the System Under Test (SUT), followed by the values *returned by the SUT*. The oracle prints back the truth values of the guarantees / modes of the original contract as comma-separated values. (How the outputs are organized depends on your system and is currently not standardized. Refer to the oracle's documentation.)

Keeping in mind a test case is a sequence of input values each corresponding to a *step* or *cycle* for the SUT, the workflow is

- read inputs `ins` for current step from the test case file
- feed it to the SUT, obtaining some outputs `outs`
- write `ins` and `outs` as comma-separated values on the oracle's standard input
- read the truth values for the original contract on the oracle's standard output

N.B. In general the values for the contract depends on previous values of the SUT's inputs / outputs. In the workflow described above, the oracle *keeps running between each*

step so that it can remember the information it needs from the previous steps to produce the next guarantee/mode truth values.

An example of a Test Execution Engine

A Test Execution Engine (TEE) compatible with Kind 2's test cases and oracles is available here:

> https://github.com/kind2-mc/teas

`Teas` is written in Python, and is able to confront a binary with Kind 2's test cases using the oracle described above. Like the Kind 2's test generation feature, `Teas` is in an experimental and unstable state.

Compilation to Rust

> While this feature has been tested on rather large systems, is still considered experimental. The Kind 2 team is eagerly awaiting feedback and bug reports to improve/fix it.

Rust is a very efficient language with a focus on safety. Kind 2 can compile Lustre to Rust, as long as the input system does not have **any unguarded pre's**, regardless of whether the initial undefined value is actually used. Arrays and records are **not** supported.

Compilation is activated by the `--compile true` flag.

The result is a Rust project in the `implem` subdirectory of the Kind 2 output directory. The project is extensively documented, you can read the documentation by running `cargo doc` in the project directory and opening `target/doc/<system>/index.html`.

Technical details

The project produces a binary that reads inputs as comma-separated values from its standard input and prints back outputs as comma-separated values on its standard output. Lustre's `real`s are compiled as 64-bits floats while `int`egers become `usize`: 32-bits (64-bits) signed integers on 32-bits (64-bits) platforms.

Assertions, properties and contracts

Compilation in Kind 2 works under the assumption that the model *has been proved correct*. Therefore properties, guarantees, and modes are not compiled as they have already been proved at model-level.

N.B. To be precise, Kind 2 works with mathematical integers and reals, not machine integers and float. Thus, it could be the case that the binary actually falsifies the specification. We are considering offering to compile properties / guarantees / modes optionally through a flag.

Assertions and assumptions from the original models are compiled as internal checks and, when falsified, will cause the binary to stop after outputting an error message pointing to the assertion / assumption falsified in the original Lustre model.

Proof Certificates

One clear strength of model checkers, as opposed to proof assistants, say, is their ability to return precise *error traces* witnessing the violation of a given safety property. Such traces not only are invaluable for designers to correct bugs, they also constitute a checkable certificate. For instance Kind 2 display a counter-example trace that shows the evolution of values of all variables in the system up to a violation of the property. In most cases, it is possible to use a counter-example for a safety property to direct the execution of the system under analysis to a state that falsifies that property. In contrast, most model checkers are currently unable to return any form of corroborating evidence when they declare a safety property to be satisfied by the system. This is unsatisfactory in general since these are complex tools based on a variety of sophisticated algorithms and search heuristics, and so are not immune to errors.

To mitigate this problem, Kind 2 accompanies its safety claims with a *certificate*, an artifact embodying a proof of the claim. The certificate can then be validated by a trusted *certificate/proof checker*, in our case the LFSC checker.

Certification chain

The certification process for Kind 2 is depicted in the graph below. Kind 2 generates two sorts of safety certificates, in the form of SMT-LIB 2 scripts: one certifying the faithfulness of the translation from the Lustre input model to the internal encoding, and another one certifying the invariance of the input properties for the internal encoding of the input system. These certificates are checked by CVC4, then turned into LFSC proof objects by collecting CVC4's own proofs and assembling them to form an overall proof that can be efficiently verified by the LFSC proof checker.

Trust is claimed at a higher level when both proof certificates are present. In practice, this means that Kind 2 didn't make any mistake in its model checking phase, and that the translation of the Lustre model to the internal representation is faithful.

Producing certificates and proofs with Kind 2

To illustrate this process, we rely on the toy model below (`add_two.lus`). The model encodes in Lustre a synchronous reactive component, `add_two`, that at each execution step other than the first, outputs the maximum between the previous value of its output variable `c` and the sum of the current values of input variables `a` and `b`. The value of `c` is initially `1.0`. The model is annotated with an invariance property stating that, at each step, the output `c` is positive whenever both inputs are.

```
node add_two (a, b : real) returns (c : real) ;
  var v : real;
```

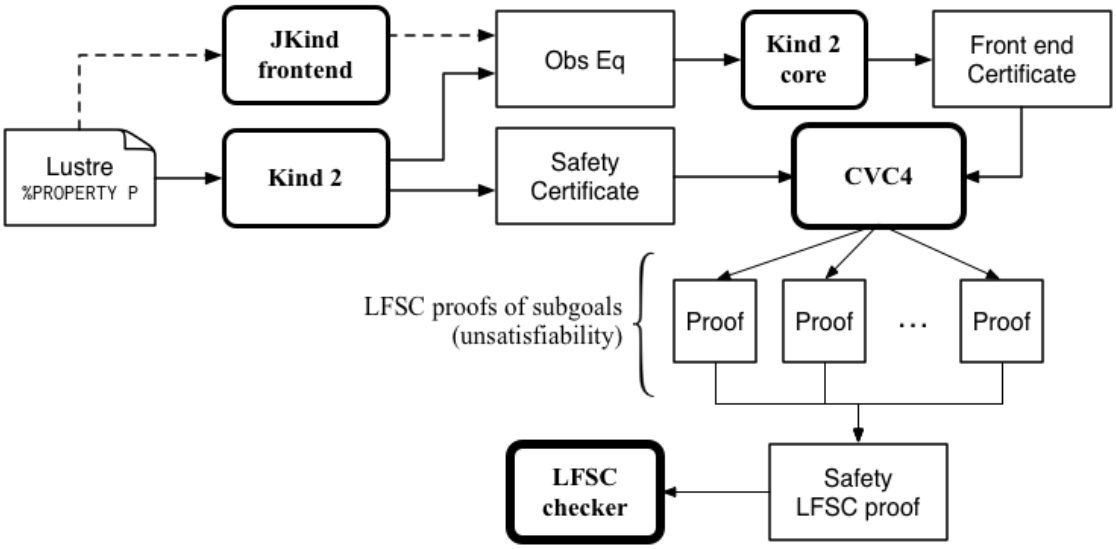

Figure 3: Certification process

```
let
  v = a + b ;
  c = 1.0 -> if (pre c) > v then (pre c) else v ;
  --%PROPERTY (a > 0.0 and b > 0.0) => c > 0.0 ;
tel
```

Kind 2 offers the possibility to generate two types of certificates, SMT-LIB 2 certificates and actual proofs in the format of LFSC. It will do so only for systems whose properties (or contracts) are all proven valid.

Requirements

Frontend certificates and proofs production require the user to have JKind installed on his machine (together with a suitable version of Java).

SMT-LIB 2 certificates do not require anything additional excepted for an SMT solver to check the certificates.

LFSC proofs production require a proof producing version of CVC4 (the binary can be specified with `--cvc4_bin`), and the LFSC checker to be compiled for the final proof checking phase.

LFSC checker The LFSC checker is also distributed with Kind 2 in the directory `lfsc`, it contains the checker and the necessary signature files with the proof rules:

```
lfsc
|-- checker
|   ...
|-- signatures
    |-- kind.plf
    |-- sat.plf
    |-- smt.plf
    |-- th_base.plf
    |-- th_int.plf
    |-- th_real.plf
```

The checker can be compiled using:

```
autoreconf -i
./configure
make
```

SMT-LIB 2 certificates

These certificates are always produced but are only used as an intermediate step for LFSC proof production. The user still has the possibility to get them as the final output of Kind 2 in a convenient form. To do so, invoke Kind 2 (on the previous example `add_two.lus`) with the following

```
kind2 --certif true add_two.lus
```

For successful runs, the output of Kind 2 will contain:

```
Certificate minimization
Kept 0 (out of 1) invariants at bound 1 (down from 1)
Certificate checker was written in add_two.out/certificates.0/certificate.smt2
Generating frontend eq-observer with jKind ...
Generating frontend certificate
...
Certificate minimization
Kept 0 (out of 4) invariants at bound 1 (down from 1)
Certificate checker was written in add_two.out/certificates.0/FECC.smt2
```

The certificates are located in the directory `add_two.out` which has the following structure:

```
add_two.out/
|-- certificates.0
```

```
|-- FEC.kind2
|-- FECC.smt2
|-- FECC_checker
|-- FECC_prelude.smt2
|-- certificate.smt2
|-- certificate_checker
|-- certificate_prelude.smt2
|-- jkind_sys.smt2
|-- jkind_sys_lfsc_trace.smt2
|-- kind2_sys.smt2
|-- observer.smt2
|-- observer_lfsc_trace.smt2
|-- observer_sys.smt2
```

In particular, it contains two scripts of interest: `certificate_checker` and `FECC_checker`. They are meant to be run with the name of an SMT solver as argument and should produce each three `unsat` results. The first one checks that the certificate of invariance is valid with the provided SMT solver and the second script checks that the *frontend certificate is valid*.

```
> add_two.out/certificates.0/certificate_checker z3
Checking base case
unsat
Checking 1-inductive case
unsat
Checking property subsumption
unsat

> add_two.out/certificates.0/FECC_checker z3
Checking base case
unsat
Checking 1-inductive case
unsat
Checking property subsumption
unsat
```

LFSC proofs

The other option offered by Kind 2, and the most trustworthy one, is to produce LFSC proofs. This can be done with the following invocation:

```
kind2 --proof true add_two.lus
```

Successful runs emit outputs that contain lines such as:

```
Certificate minimization
Kept 0 (out of 1) invariants at bound 1 (down from 1)
...
Generating frontend eq-observer with jKind ...
Generating frontend proof
...
Certificate minimization
Kept 0 (out of 4) invariants at bound 1 (down from 1)
...
Final LFSC proof written to add_two.out/add_two.lus.0.lfsc
```

The important one is the last message that indicate the file in which the proof was written. The directory produced by Kind 2 will have the following structure:

```
add_two.out/
|-- add_two.lus.0.lfsc
|-- certificates.0
    |-- FEC.kind2
    |-- base.smt2
    |-- frontend_base.smt2
    |-- frontend_implication.smt2
    |-- frontend_induction.smt2
    |-- frontend_proof.lfsc
    |-- implication.smt2
    |-- induction.smt2
    |-- jkind_sys.smt2
    |-- jkind_sys_lfsc_trace.smt2
    |-- kind2_phi.smt2
    |-- kind2_phi_lfsc_trace.smt2
    |-- kind2_sys.smt2
    |-- kind2_sys_lfsc_trace.smt2
    |-- obs_phi.smt2
    |-- obs_phi_lfsc_trace.smt2
    |-- observer.smt2
    |-- observer_lfsc_trace.smt2
    |-- proof.lfsc
```

It contains as many proofs (at the root) as there are relevant analysis performed by Kind 2 (for modular and compositional reasoning). To make sure that the proof is an actual proof, one needs to call the LFSC checker on the generated output, together with the correct signatures:

```
lfsc-checker path/to/lfsc/signatures/{sat,smt,th_base,th_int,th_real,kind}.plf add_two.ou
```

The return code for this command execution is `0` when everything was checked correctly. Two lines will be displayed when both the proof of invariance and the proof of correct translation by the frontend are valid:

```
File add_two.out/add_two.lus.0.lfsc, line 198, character 17: Check successful
File add_two.out/add_two.lus.0.lfsc, line 628, character 18: Check successful
```

In the case where only the invariance proof was produced and checked, the return code will still be `0` but only one `Check successful` will be in the output of `lfsc-checker`.

Contents of certificates

For a given problem (whose safety property is P), an internal certificate consists in only a pair `(k, phi)` where phi is a k-inductive invariant of the system which implies the original properties. SMT-LIB 2 certificates are in fact scripts whose check make sure that phi implies P and is k-inductive. The LFSC proof is a formal proof that P is invariant in the system, using sub-proofs of validity (unsatisfiability) returned by CVC4.

LFSC signature

A proof system is formally defined in LFSC through *signatures*, which contain a definition of the system's language together with axioms and proof rules. The proof system used by CVC4 is defined over a number of signatures, which are included in its source code distribution. Those relevant to this work include signatures for propositional logic and resolution (`sat.plf`); first-order terms and formulas, with rules for CNF conversion and abstraction to propositional logic (`smt.plf`); equality over uninterpreted functions (`th_base.plf`); and real and integer linear arithmetic (`th_int.plf` and `th_real.plf`).

CVC4's proof system is extended with an additional signature ('kind.plf) for k-inductive reasoning, invariance and safety. This signature also specifies the encoding for state variables, initial states, transition relations, and property predicates. State variables are encoded as functions from natural numbers to values. This way, the unrolling of the transition relation does not need the creation of several copies of the state variable tuple $\mathbf{x}$. For example, for the state vector $\mathbf{x} = (y\ ,\ z)$ with y of type real and z of type integer, the LFSC encoding will make y and z respectively functions from naturals to reals and integers. So we will use the tuples $(y(0)\ ,\ z(0))$, $(y(1)\ ,\ z(1))$, ... instead of (*y0* , *z0*), (*y1* , *z1*), ... where *y0* , *y* 1 , ..., *z0* , *z1*, ... are (distinct) variables. Correspondingly, our LFSC encoding of a transition relation formula T[$\mathbf{x}$, $\mathbf{x'}$] is parametrized by two natural variables, the index of the pre-state and of the post-state, instead of two tuples of state variables. Similarly, I, P and phi are parametrized by a single natural variable.

The signature defines several derivability judgments, including one for proofs of invariance, which has the following type:

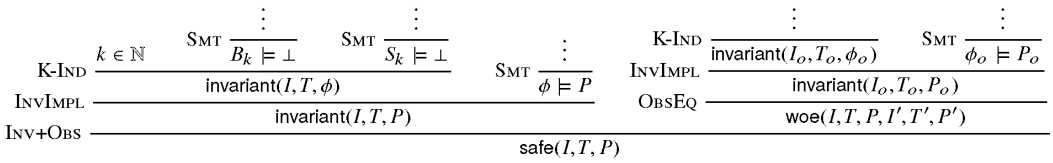

Figure 4: Proof sketch

```
invariant: Π I: ℕ → formula.
           Π T: ℕ → ℕ → formula.
           Π I: ℕ → formula. Type
```

It also contains various rules to build proofs of invariance by k-induction. This signature also specifies how to encapsulate proofs for the front-end certificates by providing a additional judgment, safe(I,T,P,I',T',P'), which can be derived only when invariant(I,T,P) is derivable and the observational equivalence between (I,T,P) and (I',T',P') is provable (judgment woe). Self contained proofs of safety follow the sketch depicted below, where Smt stands for an unsatisfiability rule whose proof tree is obtained, with minor changes, from a proof produced by CVC4.

Contract Generation

> This feature is very experimental. In particular, the modes (if any) of the contracts generated might not be exhaustive. In this case Kind 2 will reject the contract during the mode exhaustiveness check.

Contract generation is intended, at least for now, as a helper for users to getting started with Kind 2's contract language. Contract generation is activated by the flag `--contract_gen`.

Internally, this feature is implemented by running invariant generation on the input system up to some depth, specified by flag `--contract_gen_depth`. Doing so will discover equivalence and implication invariants over the system. The ones that talk only about the input / outputs of the systems are used to create the contract dumped in a lustre file in the output directory.

Invariant logging

This treatment can only run after Kind 2 concluded the system `s` is safe. If this condition is met, then invariant logging will minimize the invariants used in the proof and log them as a contract for `s` in the output directory.

> **NB**: *minimization* is understood here in terms of inclusion, not cardinality. That is, a set of invariants is *minimal* if removing an invariant either increases the `k` of the k-induction proof or causes the set of invariants to not strengthen the properties of the system any more (the properties and the remaining invariants are not k-inductive).
>
> It can be that a smaller set of strengthening invariants exists though.

The point of this feature is that it is often the case that while finding strengthening invariants takes a long time, proving the properties with these invariants is actually rather simple. Logging a minimized set of invariants allows to replay the analysis without re-discovering them.

The Kind 2 team also thinks that these invariant can turn out to be useful even after the system was modified, assuming the changes are not too important. Different invariants usually characterize different parts of the system, and some of the invariants previously logged may still apply.

Last, this feature also lets users inspect the (useful) invariants discovered by Kind 2 to make sure they conform to their understanding of the system.

The Kind 2 team is looking forward to receiving feedback on this feature, which we think can greatly improve user experience if used properly.

Failures

Logging invariants is actually rather challenging. During an analysis, the state of the whole system is made explicit. That is, Kind 2 sees the state variables of the component under analysis as well as that of all its sub-components.

Hence Kind 2 can discover invariants that relate state variables that, at Lustre level, belong to different sub-nodes burried deep in the node hierarchy. Expressing such invariants purely in terms of the top node can be challenging, especially if some node calls use `condact`s or `merge` / `activate`s.

As a consequence, the current invariant logging strategy can fail with a warning saying that it is not able to express some of the invariants at top level. If you are interested in the invariant logging feature, but run into this kind of problem, please contact us. It may be that in your case, we can solve the problem relatively easily.

Apache License

Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any

form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those

notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS