

Kind 2 User Documentation

Version v0.8.0

January 6, 2016

Contents

Kind 2	2
Requirements	2
Building and installing	3
Documentation	4
Techniques	5
Compositional reasoning	5
Modular reasoning	5
Refinement in compositional and modular analyses	5
K-induction	7
Invariant Generation	8
IC3	10
Lustre Input	11
Properties and top level node	11
Contracts	12
Arrays	18
Lustre arrays	18
Extension to unbounded arrays	20
XML Output	28
Contract semantics	29
Assume-guarantee contracts	29
Modes	30
Apache License	33

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 4.02 or later,
- Camlp4
- [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](#).

Techniques

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

- [k-induction](#)
- [invariant generation](#)
- [IC3](#)

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.

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 counterexample 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 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 = if input > max then max else input ;
```

With the optional type annotations:

```
const max : int = 42 ;
var ghost_stream : int = 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 = engage <> 0.0 ;
  var bool_dis = disengage <> 0.0 ;
  var bool_enged = engaged <> 0.0 ;

  var never_triggered = (
    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)  
) ;
```

Arrays

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.

- $A[i] = \langle \text{term}(i) \rangle$; 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, $\langle \text{term}(i) \rangle$ 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 $\langle \text{term}(i) \rangle$.
- $A = B$; 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.

$A[i] = \text{if } i = 0 \text{ then } 2 \text{ else } B[i - 1]$;

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

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

$M[i][j] = \text{if } i = j \text{ then } 1 \text{ else } 0$;

defines M as the identity matrix

$\begin{bmatrix} 1 & , & 0 & , & 0 & , & \dots & , & 0 \\ 0 & , & 1 & , & 0 & , & \dots & , & 0 \\ 0 & , & 0 & , & 1 & , & \dots & , & 0 \\ \dots & , & \dots & , & \dots & , & \dots & , & \dots \\ 1 & , & 0 & , & 0 & , & \dots & , & 1 \end{bmatrix}$

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

$$A[i] = \text{if } i = 0 \text{ then } 0 \text{ else } A[i-1] ;$$

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 $A[i] = \text{pre } A[i]$; 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 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 $A[k]$.

$$\begin{aligned} A[k] &= B[k+1] + y; \\ B[k] &= C[k-1] - 2; \\ C[k] &= A[k] + k; \end{aligned}$$

On the other hand this one will be accepted.

$$\begin{aligned} A[k] &= B[k+1] + y; \\ B[k] &= C[k-1] - 2; \\ C[k] &= (A[k-1] + B[k]) * k ; \end{aligned}$$

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 homomorphic 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
<code>--smt_arrays</code>	Use the builtin theory of arrays in solvers
<code>--inline_arrays</code>	Instantiate quantifiers over array bounds in case they are statically known
<code>--arrays_rec</code>	Define recursive 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

$$\forall k: \text{int}. 0 \leq k < 6 \Rightarrow (\text{select } A \ k) = x$$

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

`--smt_arrays {#n11902657---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 {#n11902657---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

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

--arrays_rec {#n11902657---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.

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

$$(\Box A) \Rightarrow (\Box G)$$

where \Box 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 $(\{a_1, \dots, a_n\}, \{g_1, \dots, g_m\})$ 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

$$G' = G \cup \{ (\bigwedge r_i \Rightarrow (\bigwedge e_i), (\{r_i\}, \{e_i\}) \text{ in } M) \}$$

where \cup 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

$$\text{one_mode_active} = \bigvee \{ r_i \}$$

is an invariant for the system

$A \wedge G \wedge \{ r_i \Rightarrow e_i \}$

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.

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

Copyright {2015} {Board of Trustees of the University of Iowa}

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.