Kind 2 User Documentation

Version 3.0.0

October 14, 2026

1 Kind 2 1

1.1 Try Kind 2 Online 2
1.2 Download 2
1.3 Required Software 2
1.4 VS Code Extension 3
1.5 Docker 3
1.6 Building and installing 5
1.7 Development 7
1.8 Documentation 7

2 Techniques 8

2.1 Compositional reasoning 8
2.2 Modular reasoning 8
2.3 Refinement in compositional and modular analyses 9
2.4 Modifiers to control node/function abstraction 10

3 k-Induction 11

3.1 Options 11

4 Invariant Generation 13

4.1 Options 13
4.2 Lock Step K-induction 14

5 IC3 15

5.1 IC3-IA Options 16
5.2 IC3-QE Options 16

6 Kind 2 Input 17

6.1 Properties and top-level node 17
6.2 Contracts 21
6.3 Underspecified outputs 30
6.4 Theimportedkeyword 31
6.5 Functions 32

6.6 Lemmas 40
6.7 Integer division and modulo 44
6.8 Thefbyoperator 45
6.9 Conditional expressions 45
6.10 Short-circuit Boolean operators 46
6.11 If statements and frame conditions 47
6.12 Polymorphic nodes 58
6.13 Polymorphic contracts 61
6.14 Nondeterministic choice operator 63

7 Arrays 66
7.1 Lustre arrays 66
7.2 Extension to unbounded arrays 72

8 Machine Integers 79
8.1 Declarations 79
8.2 Values 79
8.3 Semantics 80
8.4 Operations 81
8.5 Limitations 83

9 Refinement Types 84
9.1 Declarations 84
9.2 Semantics 85
9.3 Operations 86
9.4 Type Ascription 86
9.5 Realizability 87
9.6 Constants 88
9.7 Structured types 88

10 Enumeration types 90
10.1 Selecting on an enumerated value 90

11 History Types 91

12 Abstract Types 92
12.1 Domain and quantification 92

13 Polymorphic User Types 94

14 Tuples 95
14.1 Element update 95

15 Sets 97

16 Maps 99

17 Subrange types 101

17.1 Symbolic bounds 101

18 Records 103

18.1 Element update 103

19 Algebraic Datatypes 104

19.1 Declarations 104

19.2 ADT Terms 104

19.3 Pattern Matching 105

19.4 Testers 106

19.5 Selectors 106

19.6 Polymorphic ADTs 107

20 JSON / XML Output 108

20.1 JSON format 108

20.2 XML format 114

21 Exit codes 119

21.1 Code Convention 119

21.2 Former Convention 120

22 Contract Semantics 121

22.1 Assume-guarantee contracts 121

22.2 Modes 122

23 Post Analysis Treatments 126

23.1 Prerequisites 127

24 Test Generation 128

24.1 Combinations of modes as abstractions 128

24.2 Generating test cases 131

24.3 Analyzing the executable 132

25 Proof Certificates 133

25.1 Certification chain 133

25.2 Producing certificates and proofs with Kind 2 134

25.3 Contents of certificates 139

25.4 CPC signature 139

26 Contract Generation 141

27 Invariant Printing 142

28 Interpreter 143
28.1 Structure of the input file 143
28.2 Integers and reals 144
28.3 Records 144
28.4 Arrays 144
28.5 Tuples 145

29 Contract Monitor 146

30 Inductive Validity Core 149
30.1 Options 149
30.2 Example 150
30.3 Minimizing over a subset of the assumptions/guarantees 151
30.4 Computing all Inductive Validity Cores 152

31 Minimal Cut Set 153
31.1 Options 153
31.2 Example 154

32 Contract Check 156

33 Assumption Generation 157

34 A Lucent Primer 159
34.1 Basic Concepts 159
34.2 Comments 162
34.3 Primitive Types 162
34.4 Conditionals 163
34.5 Temporal Operators 164
34.6 Frame Blocks 167
34.7 Declarative Semantics 168
34.8 Composite Types 169
34.9 Enumerations and Subranges 172
34.10 Refinement Types 173
34.11 Abstract Types 174
34.12 Composition 174
34.13 Lazy Operators 178
34.14 Nondeterministic Choice 181
34.15 Type Ascription 181
34.16 Common Auxiliary Nodes 181
34.17 Specifying and Checking Properties 182
34.18 Contracts 184

34.19 Compositional and Modular Analysis 186
34.20 Realizability Checking 187
34.21 More Examples 188

35 Apache License 189

1 Kind 2

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

Kind 2 is a command-line tool. It takes as input a Lustre file annotated with properties to be proven invariant (see Kind 2 Input), 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 external tools, Kind 2 can output its results in JSON and XML formats (see JSON / XML Output).

By default Kind 2 runs a process for bounded model checking (BMC), two processes for k-induction (one for a fixed value of k=2, and other for increasing values of k), several processes for invariant generation, a process for IC3QE, and several processes for IC3IA 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 Techniques for configuration examples and more details on each technique.

`--enable {BMC|IND|IND2|IC3QE|IC3IA|INVGEN|INVGENOS|...}` Select model checking engines

By default, all five model checking engines are run in parallel. Give any combination of `--enable BMC`, `--enable IND`, `--enable IND2`, `--enable IC3QE` and `--enable IC3IA` to select which engines to run. The option `--enable BMC` alone will not be able to prove properties valid, choosing `--enable IND` and `--enable IND2` only (or either of the two alone) 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 <int>` (default `0` = none) – Run for the given number of seconds of wall clock time

`--smt_solver {Bitwuzla|cvc5|MathSAT|OpenSMT|SMTInterpol|Yices2|Z3}` (default `Z3`) – Select SMT solver

`--bitwuzla_bin <file>` – Executable for Bitwuzla

`--cvc5_bin <file>` – Executable for cvc5

`--mathsat_bin <file>` – Executable for MathSAT 5

`--opensmt_bin <file>` – Executable for OpenSMT

`--smtinterpol_jar <file>` – JAR of SMTInterpol

`--yices2_bin <file>` – Executable for Yices 2 (SMT input)

`--z3_bin <file>` – Executable for Z3

`-v` Output informational messages

`-json` Output in JSON format

`-xml` Output in XML format

1.1 Try Kind 2 Online

Visit our web interface to try Kind 2 from your browser.

1.2 Download

If you use a Linux or a macOS computer, you can download an executable of the latest version of Kind 2 from here. First make sure though that you have the required software described next.

1.3 Required Software

To run Kind 2 the following software must be installed on your computer:

- Linux or macOS, and
- a supported SMT solver
  - Bitwuzla (for inputs with only machine integers),
  - cvc5,
  - MathSAT 5,
  - OpenSMT (v2.8.0),
  - SMTInterpol,
  - Yices 2, or

- Z3

Z3 is the presently recommended SMT solver and the default option. For best results, we recommend using a combination of several solvers. For systems with integer and real variables, we recommend using Z3 as the main solver (`--smt_solver Z3`) and MathSAT as the interpolating solver (`--smt_itp_solver MathSAT`). For systems with only machine integers, we recommend using Bitwuzla as the main solver (`--smt_solver Bitwuzla`), MathSAT as the interpolating solver (`--smt_itp_solver MathSAT`), and Z3 for performing quantifier elimination (`--smt_qe_solver Z3`).

1.4 VS Code Extension

You can also install our extension for Visual Studio Code which provides support for Kind 2. The extension contains Linux and macOS binaries for Kind 2 and Z3 ready to use. Windows is also supported through WSL2 (see here for more details).

1.5 Docker

Kind 2 is also available on Docker Hub.

1.5.1 Retrieving / updating the image

Install docker and then run

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

Docker will retrieve the *layers* corresponding to the latest development version of the Kind 2 repository (`main` branch). 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.

1.5.2 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 development version (`main` branch);
- `docker run` does **not** update your local Kind 2 image to the latest one: the appropriate `docker pull` command does.

1.5.3 Packaging your local version of Kind 2

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

```
docker build -t kind2-local -f ./docker/Dockerfile .
```

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 cvc5 solvers.

1.6 Building and installing

If you prefer, you can build Kind 2 directly from sources, either through the OPAM package manager (recommended) or directly using dune.

1.6.1 Using OPAM

Start by installing OPAM 2.x following the instructions on the website, and make sure OPAM has been initialized by running `opam init`. If you want to build the development version of Kind 2 that includes the most recent changes, as opposed to the latest release, then run

```
opam pin add -n kind2 https://github.com/kind2-mc/kind2.git
```

(You can always undo this change later using this command `opam unpin kind2`).

Otherwise, skip the step above and either run

```
opam install --update-invariant kind2
```

if you have OPAM 2.1 or later installed on your system, or run

```
opam depext kind2
opam install --unlock-base kind2
```

if you have an older version of OPAM (you can run `opam --version` to check the version).

This builds and installs a compatible version of the OCaml compiler and libraries, and the `kind2` binary. Now you can start using `kind2`.

Other options using OPAM

By default, `kind2` will be installed into the bin directory of your current OPAM switch. Run

```
opam install kind2 --destdir=<DIR>
```

to install the Kind 2 binary into `<DIR>/bin`. This will also create directories `<DIR>/doc` and `<DIR>/lib`.

In alternative, you can clone https://github.com/kind2-mc/kind2.git, move to its top-level directory, and run

```
make install
```

to have OPAM install `kind2` and its dependencies.

Note that z3 is available in OPAM so it is possible to install it too with OPAM by running:

```
opam install z3
```

Be aware, however, that this takes quite a bit of time (up to 25 minutes).

1.6.2 Direct Installation Using Dune

To build directly from sources you will also need the following software first:

- OCaml 5.5 or later,
- Dune 2.7 or later,
- dune-build-info,
- Yojson,
- num,
- Menhir parser generator

First install this software on your system using your preferred method. Then clone the Kind 2 git repository, move to the top-level directory of the repository, and run

```
dune build src @install
dune install --sections=bin --prefix <DIR>
```

to install the Kind 2 binary into `<DIR>/bin`.

You need a supported SMT solver in your PATH environment variable when running `kind2`.

1.7 Development

With OPAM 2.x you can create a local switch which will install all dependencies automatically.

```
opam switch create .
make
```

Alternatively, you can install all dependencies in your current switch by running:

```
opam install . --deps-only
make
```

For running the unit tests for front end, you can install ounit2 library using opam by running:

```
opam install ounit2
```

To run the ounit tests, you can use the following dune command:

```
dune test
```

To run regression tests, you need a Python interpreter and the pytest module. The simplest way to do this is to install uv. The test-running script will try to use `uv` to set up its own isolated python environment with `pytest`, so the rest of your system will not be impacted. If it fails to find `uv`, it will fallback to trying to use `pytest` in your system's Python interpreter.

```
make test
```

1.8 Documentation

Documentation is available online in HTML or PDF forms.

See `doc/content/README.md` for more information.

2 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

2.1 Compositional reasoning

When verifying a node `n`, *compositional reasoning* consists in abstracting the complexity of the subnodes of `n` by their contracts (see Contract Semantics). 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 (`--compositional true`), Kind 2 will abstract all calls (to subnodes that have a contract with at least one guarantee or one mode) 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.

2.2 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. The analysis is completed on every node even if some node is proved unsafe because of the falsification of one of its properties.

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.

2.3 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.

A recursive function is refined the same way, in steps. In a compositional analysis its contract stands in for its recursive calls; once its own analysis has proved that contract, refining a call to it from another node or function first unrolls the body of the function once, with its recursive calls abstracted by the contract, then up to `--rec_unrollings` times (2 by default); the values of the calls that a spurious counterexample relied on are given to the refinements as well, so that they need not unroll the function as deep as the calls go. The analysis of the function itself, or of a function of its recursive group, is not refined this way: there the contract remains the induction hypothesis of the recursion.

2.4 Modifiers to control node/function abstraction

To prevent Kind 2 from *abstracting* a specific node or function that has both a body and a contract during compositional analysis, use the `transparent` modifier before the `node` or `function` keywords:

```
transparent function F(...) returns (...)
```

To prevent Kind 2 from *refining* a specific node or function that has both a body and a contract during compositional and modular analyses, use the `opaque` modifier before the `node` or `function` keywords:

```
opaque node N(...) returns (...)
```

3 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>
```

3.1 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.

4 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.

4.1 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 one-state invariants implied by previous one-state invariants, and two-state invariants implied by previous two-state invariants or 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.

4.2 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.

5 IC3

Kind 2 supports two SMT-based extensions of the SAT-based verification technique IC3. The challenge when lifting IC3 to infinite state systems is the computation of pre-images of the system's transition relation. The first extension, IC3QE, uses quantifier elimination to compute the pre-image. If the input problem is in linear integer arithmetic, Kind 2 uses a built-in method that performs a fast approximate quantifier elimination. Otherwise, the quantifier elimination is delegated to an SMT solver, which is at this time possible with Z3 and cvc5.

The second extension, IC3IA, implements a version of IC3 that relies on implicit (predicate) abstraction by Cimatti et al. The main idea of the method is to work on an abstraction of the state space induced by a set of predicates so that the computation of pre-images of the system's transition relation does not require the use of quantifier elimination. As with most abstraction methods, this introduces the problem of having to handle spurious abstract counterexamples for the property to be proven, that is, traces that falsify the property in the abstracted system but are not actual executions of the original system. The method addresses this problem by using an abstraction refinement technique based on logical interpolants. Kind 2 currently supports three proof-based interpolating SMT solvers for the generation of interpolants: MathSAT, SMTInterpol, and OpenSMT. In addition, Kind 2 also implements a built-in method for producing interpolants based on quantifier elimination that can be used with Z3 and cvc5. When the IC3IA engine is enabled, each property is handled separately by a different process; two when the built-in method for producing interpolants is used, one generating backward interpolants and another one generating forward interpolants.

To enable nothing but the IC3 engines (IC3QE and IC3IA), run

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

If you only want to enable of the engines, e.g. IC3QE, run

```
kind2 --enable IC3QE <file>
```

5.1 IC3-IA Options

`--smt_itp_solver {MathSAT | SMTInterpol | Z3qe | cvc5qe | OpenSMT | Bitwuzla}` – set the SMT solver used for interpolation.

`--ic3ia_max <int>` – set the maximum number of IC3IA parallel processes. Each process checks an individual property.

5.2 IC3-QE Options

`--qe_method {precise|impl|cooper}` (default `cooper`) – select the quantifier elimination strategy: `cooper` for the built-in approximate method, `precise` or `impl` to delegate to the SMT solver. The `precise` strategy computes the exact pre-image, which is an expensive operation. The `impl` strategy under-approximates the result by computing a conjunctive implicant first. If the problem is not in linear integer arithmetic, `cooper` falls back to `impl`.

`--smt_qe_solver {Z3 | cvc5}` (default `detect`) – set the SMT solver used for quantifier elimination

To see other advanced options run `--help_of ic3qe`.

6 Kind 2 Input

Kind 2 reads input models written in an extension of the dataflow Lustre language (see this primer for a quick introduction to the Lustre 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.

6.1 Properties and top-level node

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

```
--%PROPERTY ["<name>"] <bool_expr> ;
```

or, use a `check` statement:

```
check ["<name>"] <bool_expr> ;
```

where `<name>` is an identifier for the property and `<bool_expr>` is a Boolean Lustre expression.

In addition to invariant properties, Kind 2 also accepts dedicated syntax for checking the existence of a witness. You can specify reachability properties of the form:

```
--%PROPERTY reachable ["<name>"] <bool_expr> [from <int>] [within <int>];
```

or, using a `check` statement:

```
check reachable ["<name>"] <bool_expr> [from <int>] [within <int>];
```

where the clauses between square brackets are optional. The optional clauses allow you to specify, exclusively or at the same time, a lower and upper bound on the number of execution steps in the witness trace. Concretely, `check reachable P from m` asks whether a state satisfying `P` is reachable in `m` steps or more while `check reachable P within n` asks whether a state satisfying `P` is reachable in `n`

steps or less. Moreover, Kind 2 also supports the following syntax for the specification of properties where the lower and upper bounds are the same:

```
check reachable ["<name>"] <bool_expr> at <int>;
```

Without modular reasoning active, Kind 2 only analyzes the properties of what it calls the *top nodes*. By default, any node that is not depended on by another node (i.e. called by that node) is a top node. Alternatively, nodes can be marked as *main nodes* by doing the following:

```
--%MAIN ;
```

to the body of that node.

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

```
kind2 --lus_main <node_name> ...
```

Main nodes specified by the command line option override main nodes annotated in the source code. If any main nodes exist then only main nodes are analyzed (top nodes are not).

6.1.1 Examples

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 a *main 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 v1.5.1

================================================================
Analyzing top
  with First top: 'top'
              subsystems
                | concrete: intcounter, greycounter

<Success> Property OK is valid by inductive step after 0.065s.

----------------------------------------------------------------
Summary of properties:
----------------------------------------------------------------
OK: valid (k=5)
================================================================
```

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

The second example demonstrates reachability properties using a single `counter` node:

```
node counter () returns (out: int);
let
   out = 0 -> pre out + 1;

   check reachable out = 10;
   check reachable out = 100 from 99;
   check reachable out = 50 at 50;
   check reachable out = 15 from 10 within 20;

   check reachable out = 10 within 5;
tel
```

Kind 2 produces output reporting that the first four expressions are reachable, while the last is not. If you want to print a witness in the standard output for each proven reachability property, pass `--print_witness true` to Kind 2. To dump the witness to a file instead, pass `--dump_witness true` to Kind 2.

6.1.2 Conditional Properties

Invariant properties of a node are often case-based, with each case describing what the component should do depending on a specific situation. These properties are usually encoded in conditional properties of the form `situation => behavior`, and are often better represented in terms of the mode logic of a node (see subsection Modes in Contract Semantics). However, these properties do not always imply modal behavior, or they are not defined in terms of the interface of a node. For those cases, Kind 2 allows the user to specify a conditional invariant property of the form `B => A` as follows:

```
check A provided B;
```

This dedicated syntax makes writing properties more straightforward and user-friendly, but also allows Kind 2 to trigger additional checks. A challenge for the user with these kinds of properties arises if the guard `B` may always be false, for example due to a modeling error. The user may believe that the property is interesting and true, whereas the property is vacuously true.

When the dedicated syntax above is used, Kind 2 simultaneously checks that `B => A` is invariant and `B` is reachable. If `B => A` is in fact invariant, the reachability check lets you know whether the implication is trivially true or not. Notice that when running Kind 2 in modular mode, the reachability check is performed locally

to a node without taking call contexts into account; only the specified assumptions are considered. You can disable this check by passing `--check_nonvacuity false` to Kind 2, or by suppressing all reachability checks (`--check_reach false`).

6.2 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.

6.2.1 Inline syntax

A local contract is a block 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 block is denoted by the keywords con and `noc`:

```
con
  [item]+
noc
```

The original contract syntax (which is deprecated but still available) is a special block comment of the form

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

or

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

6.2.2 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.

6.2.3 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 depend on 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 depend on. 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`. 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. However, each contract import introduces its own namespace to avoid name collisions.

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`, the actual input parameters of the import of `c` cannot depend on outputs of `n` in the current state. The reason is that the distinction between inputs and outputs lets Kind 2 check that the assumptions requirements make sense, *i.e.* do not depend on outputs of `n` in the current state.

The general syntax is

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

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
) ;
con
  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) ;
noc
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 **outside** the mode item itself. That is, no self-references are allowed. Forward 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)
) ;
```

6.2.4 Restart

Kind 2 supports resetting the internal state of an expression to its initial state with the construct restart/every. Writing

```
restart e every r
```

evaluates the expression `e`, and every time the Boolean stream `r` is true, the internal state of `e` is reset to its initial state: every `->`, `pre` and `fby` operator in `e`, and every node called in `e`, starts again as at the first step. The reset happens at the very step at which `r` is true, so the value of `e` at that step is computed from its initial state. The condition `r` itself is not reset, and neither are the streams `e` reads from outside: only the state that `e` holds is.

In the example below, the node `top` calls `counter` (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(3) every reset;
tel
```

A trace of execution for the node `top` could be:

step	reset	c
0	false	0
1	false	1
2	false	2

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

The restarted expression does not need to be a node call: `x = restart (0 -> pre x + 1) every reset;` defines the same counter, without the modulo. The arguments of a call are part of the restarted expression, so in `restart counter(n) every r` any state that `n` holds is reset as well. An argument whose state must not be reset is defined outside the restart, in a local variable.

Like `if ... then ... else ...`, the condition `every r` extends as far to the right as possible: `restart e every r or s` is `restart e every (r or s)`.

A group of equations can be restarted together with a restart block:

```
restart
  s = 0 -> pre s + i;
  m = 0 -> if s > pre m then s else pre m;
every reset end
```

All the streams defined in the block, and the state of the expressions that define them, are reset every time `reset` is true. A restart block may contain equations, assertions, and `if`, `when`, `match` and restart blocks, but no properties, main annotations or frame blocks.

> **Note:** 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 clocks. A restart inside a branch of a `when` expression or block (see Conditional expressions) is evaluated on the clock of the branch: in

```
o = when c then (restart counter(3) every r) else -1;
```

`counter` is only reset at the steps at which both `c` and `r` are true, and a step at which `r` is true while `c` is false has no effect. A restart around a `when` expression, on the other hand, also resets the state of the branches that are not selected: in

```
o = restart (when c then counter(3) else -1) every r;
```

`counter` starts again from its initial state the next time `c` is true after any step at which `r` was true.

Restrictions. A restart cannot appear in a function or in the definition of a constant, since neither has state. The `last` operator cannot be used under a restart, and a restart cannot apply to values whose type is a type parameter of a polymorphic node.

Clock operators. Kind 2 does not support the clock operators `merge`, `when` (as a sampling operator, `e when c`), `current`, `activate` and `condact` of Lustre and Scade. Their behavior is obtained with `when` expressions, which evaluate only the selected branch, and with restart:

Clock operator	With `when` and `restart`
`merge c (true -> e1 when c) (false -> e2 when not c)`	`when c then e1 else e2`
`merge k (A -> e1 when A(k)) (B -> e2 when B(k)) (C -> e3 when C(k))`	`when k = A then e1 else when k = B then e2 else e3`
`merge c (true -> (activate N every c)(x)) (false -> e when not c)`	`when c then N(x) else e`
`condact(c, N(x), d)`	`o = when c then N(x) else h; h = d fby o;`
`condact(c, (restart N every r)(x), d)`	`o = when c then (restart N(x) every r) else h; h = d fby o;`
`(restart N every r)(x)`	`restart N(x) every r`

A case analysis on more than two values, such as a `merge` on an enumerated clock, reads more naturally as a cond block, which has the same meaning as the nested `when` expressions:

```
cond
  | k = A: o = e1;
  | k = B: o = e2;
  otherwise: o = e3;
end
```

Temporal operators and node calls in a branch of a `when` expression run on the clock of the branch: their state only advances at the steps at which the branch is selected. A sampled expression `e when c` was evaluated at every step, and only its value was selected, so an `e` with state of its own is defined outside the branches. For the same reason, the previous value `h` that `condact` keeps while `c` is false is defined outside the `else` branch, where `fby` would run on the clock `not c`. The regression test clock_operators_with_when_and_restart.lus checks each of these implementations against a reference.

6.3 Underspecified outputs

Every output (and local variable) of a node or function must be defined in its body, either by an equation or by a frame block; leaving an output without a definition is rejected. (The only exception is `imported` nodes and functions, which have no body at all; see The imported keyword.)

An output need not be given a precise value, however. It can be left *underspecified* by assigning it an arbitrary value of the appropriate type with the `any` (or `choose`) operator (see Nondeterministic choice operator). This is the intended way to express that an output is not fully constrained. For instance, the node below defines `count` precisely but leaves `error` underspecified, while still being analyzed against its contract:

```
node count (trigger: bool) returns (count: int ; error: bool) ;
con
  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 ;
  ) ;
noc
let
  count = (if trigger then 1 else 0) + (0 -> pre count) ;
  error = any@<bool> ;
tel
```

This node can be analyzed: first for mode exhaustiveness, and then the body is checked against its contract. Here, both will succeed.

6.4 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

```
con
  assume true ;
  guarantee true ;
noc
```

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. Every output of an imported node is assumed to depend on every input. This may lead Kind 2 to detect circular dependencies that do not exist in an _actual system, resulting in the rejection of an input model. To make Kind 2 accept

such model, the imported node must be refined by decomposing it into smaller subnodes and specifying the actual dependencies among inputs and outputs.

6.5 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` must be a *non-temporal* combination of its inputs. That is, a function cannot depend on the `->`, `pre`, `fby` or `restart` operators. A function is also not allowed to call a node, only other functions. In Lustre terms, functions are stateless.

In Kind 2, these restrictions also apply to the contract attached to a function, if any. Moreover, Kind 2 strictly enforces that imported functions and functions abstracted by their contracts behave as mathematical functions. That is, given the same inputs, such a function always produces the same outputs, regardless of the step at which it is called.

The stateless nature of functions also determines the scope of their contract assumptions. For a function, an assumption constrains only the *current* timestep: when reasoning about a call, the function's guarantees may rely on its assumptions holding at the current step alone. For a node, by contrast, the scope extends to all previous timesteps: the node's guarantees may rely on its assumptions having held at every step up to and including the current one (the "assumptions always hold implies guarantees always hold" semantics described in Contract Semantics). This mirrors the fact that a function's outputs depend only on the current values of its inputs, whereas a node may also depend on their previous values.

6.5.1 Recursive functions

A function may call itself, directly or through a cycle of other functions, if it is declared with the `rec` modifier:

```
datatype Nat = Succ (pred: Nat) | Zero;

function rec to_int (n: Nat) returns (out: int)
con
  decreases n;
noc
let
  out = match n with
    | Zero      : 0
```

```
    | Succ (m)  : 1 + to_int (m)
  end;
tel
```

Every function marked `rec`, and every function reachable from it through a call cycle, must carry a `decreases` contract item. This measure is what lets Kind 2 establish that the recursion terminates (without it, a call could be given a definition that has no solution). Kind 2 rejects a `rec` function that lacks a `decreases` clause, and rejects a plain (non-`rec`) function that is found to actually be part of a (recursive) call cycle. A lemma needs a `decreases` clause only when it invokes itself, directly or through a cycle of other lemmas; a lemma without such a call may omit it.

A `decreases` clause is only meaningful in the inline contract of a `rec` function or of a lemma, and exactly one clause is allowed there. Declaring one anywhere else is an error.

A `decreases` clause takes one of two forms. In either form, the measure may only mention the input parameters of the function and constants. It may call a non-recursive function that can be inlined: one with no contract with guarantees, modes or a refinement type on an output (or declared `transparent`), with a single output, whose body is made of equations that define its output and local variables one at a time, without assertions or variables of an array type, and which calls only functions that can be inlined in turn. The call is replaced by the body of the function, and the obligations of the call, such as the assumptions of the function, are checked as for any other call. A type ascription is allowed as well. A measure cannot contain a `choose` or `any` operator, or call a node, a recursive or imported function, or a function with a contract that abstracts it; a measure that needs a value computed by such a function can take that value as an additional input parameter.

Integer measure. A single integer expression, or a comma-separated tuple of integer expressions read lexicographically:

```
type Count = subrange [0,*] of int;

function rec sum_to (n: Count) returns (out: int)
con
  decreases n;
noc
let
```

```
  out = when n = 0 then 0 else n + sum_to (n - 1);
tel
```

For an integer measure, Kind 2 generates two proof obligations per recursive call and verifies them like any other property: the measure must be bounded below by `0`, and it must strictly decrease (lexicographically, for tuples) from caller to callee. The analysis of the function itself, in a modular analysis (`--modular true`), verifies them for every recursive call of the function, whether or not a property or a guarantee depends on the call, and whether or not the function has a contract.

Note the use of `when ... then ... else` rather than plain `if ... then ... else`: in Lustre, both branches of an `if` are part of the expression's definition regardless of which one is selected, so the `else` branch's `sum_to (n - 1)` would still need to be well-defined (and its argument still in range) even when `n = 0`. `when ... then ... else` guards the untaken branch instead, so the recursive call is only ever made with `n - 1`, which stays within `Count` precisely because it is guarded by `n = 0` being false. This is also why the input is restricted to `Count`: the measure must be bounded below by `0` for the recursion to be well-founded, and this only holds for non-negative `n`.

Algebraic-datatype measure. A single expression whose type is a recursive ADT (see Algebraic Datatypes). This form cannot be used as a component of a tuple measure. Instead of generating a property, Kind 2 checks ADT measures **statically**, at compile time: for every recursive call, the callee's measure (after substituting the actual call arguments for the callee's parameters) must be a variable that the caller obtained by pattern-matching — possibly through several nested matches — on its own measure. Concretely: matching on the measure itself (or on a variable already known, from an earlier match, to be such a variable) and binding one of the resulting constructor's fields makes that field's variable an accepted witness that the recursion is decreasing. For example:

```
datatype Nat = Succ (pred: Nat) | Zero;

function rec is_even (n: Nat) returns (b: bool)
con
  decreases n;
noc
let
  b = match n with
    | Zero      : true
    | Succ (m)  : is_odd (m)
```

```
  end;
tel

function rec is_odd (n: Nat) returns (b: bool)
con
  decreases n;
noc
let
  b = match n with
    | Zero      : false
    | Succ (m)  : is_even (m)
  end;
tel
```

Here `is_even`'s call to `is_odd (m)` passes `m` in the position of `is_odd`'s own measure `n`. The match arm `Succ (m)` is matching directly on `is_even`'s measure `n`, so `m` — the field it binds — is accepted as a witness that the call decreases. The same reasoning applies to `is_odd`'s call back into `is_even`, so the mutual recursion is accepted as terminating.

Because a match's tester (e.g. `Succ?(n)`) is what actually guarantees a pattern-bound variable is a genuine substructure of the matched value, only such variables are accepted. A **raw field selector is never accepted**, even when applied to the exact same field a match would have bound, and even when it appears directly guarded by an `if`/`when` on the right tester (e.g. `if Succ?(n) then is_even (n.pred) else ...`): applied to the wrong constructor a selector is unconstrained, and the checker has no way to confirm, from an `if`/`when` alone, that the guard actually holds at the call. Likewise, an argument computed through an intermediate local variable, an auxiliary call, or any other indirection is rejected even if it is semantically equal to a directly pattern-matched variable.

The check also applies to calls written inside a type annotation — a refinement predicate on an input, output or local, or an array bound. Such a call can never be decreasing, since only a match in the function's body can witness a decrease, so it is always rejected.

6.5.2 How recursive functions are analyzed

Kind 2 unrolls the definition of a recursive function a bounded number of times: a call to the function is expanded to its body, and so are the recursive calls inside that body, up to a number of unrollings; what the recursive calls past those unrollings stand for depends on the kind of analysis.

By default, and in a modular analysis (`--modular true`), such a call is left unconstrained: its outputs are tied to the outputs of the other calls of the function with the same arguments, and to nothing else. A property that holds of the unrolled function then holds of the function, while a counterexample that reaches such a call may be spurious. Kind 2 then evaluates the calls the counterexample relies on, which are at concrete arguments, with the definition of the function, and reports the counterexample only if the function as it is violates the property with the same inputs; the property itself, or what the path relies on, such as an assumption on the inputs, may depend on the calls. A function that reads a global constant is evaluated with the value the counterexample gives the constant. Otherwise, or if the evaluation takes too long, Kind 2 unrolls the function once more, from one unrolling up to the limit set with `--rec_unrollings` (2 by default), and runs its engines again on the new system, keeping what they had established; this is not an analysis of its own, and it stops as soon as the arguments of the recursive calls are exhausted, as for `Fact(4)`, which is proved equal to 24 after five unrollings, with `--rec_unrollings 5`. In a modular analysis, such a call at constant arguments is evaluated instead (see below), whatever the limit. A property whose counterexample still reaches such a call at the limit is left unknown, and Kind 2 says so. That is the fate of a property that holds of the function for every input only by induction over the recursion, such as `Fact(n) > 0`: a compositional analysis, or a lemma, is what proves it. The termination checks are verified as properties at every unrolling. A function whose body makes several recursive calls, or calls other recursive functions, has its instances multiplied at every unrolling: the Ackermann function has three times as many after each one. The unrolling stops early when the next one is expected to exceed the number of instances set with `--rec_instances` (100 by default), and the properties whose counterexamples reach the recursive calls are left unknown as at the limit. An `opaque` function is the exception: its recursive calls past the unrollings are abstracted by its contract in every analysis, as described next for a compositional one. A lemma is opaque, which is what lets its guarantees be proved by induction over its recursion.

In a compositional analysis (`--compositional true`), the recursive calls past the unrollings of a function that has a contract are abstracted by that contract instead: Kind 2 assumes their guarantees, the inductive hypothesis of the recursion, which is only justified when the termination checks hold. This is what makes a property such as `Fact(n) > 0` provable from a `guarantee f > 0`, but it also means that the function is essentially unknown to the solver past its unrollings: `Fact(4) = 24` cannot be established that way. A function with no contract to abstract it with (no guarantee and no mode with an ensure, whether explicit or coming from a refinement

type on an output; assumptions and input types do not count, they are obligations of the callers), or declared `transparent`, is unrolled with its recursive calls left unconstrained, as above, while an `opaque` function is always abstracted by its contract.

In the analysis of the function itself, which proves its contract, the body is unrolled once before the recursive calls are abstracted by the contract. A guarantee may hold without following from the guarantees of the calls one level down: `Alt(n) = 1 - Alt(n - 1)` with `Alt(0) = 0` only takes the values 0 and 1, so `guarantee r >= 0` holds, but assuming it of `Alt(n - 1)` allows `Alt(n - 1) = 2` and `Alt(n) = -1`. Two levels down, `Alt(n) = Alt(n - 2)`, and the guarantee follows. The option `--rec_contract_unrollings` sets how many times the body is unrolled in the analysis of the function itself (1 by default); a stronger contract, such as `0 <= r and r <= 1` here, is the alternative.

When the analysis is both compositional and modular, a call to a recursive function from another node or function is refined like any other call (see refinement), in steps: if the contract of the function is not enough to prove the properties of the caller, and the analysis of the function itself proved its contract valid, the caller is analyzed again with the body of the function unrolled once, its recursive calls abstracted by the contract; if that is not enough either, with the body unrolled twice, and so on up to the limit set with `--rec_unrollings`. This refinement does not apply to the analysis of the recursive function itself, or of a function of its recursive group: there the recursive calls keep the contract as their induction hypothesis.

A counterexample found with a recursive function abstracted by its contract is checked against the function as it is: Kind 2 evaluates the calls of the function the counterexample executes, at their concrete arguments, and checks whether the property still fails with their values, on the system sliced to the property. If it does, the property is false of the function itself, and unrolling the function, which only replaces its contract by its body, cannot prove it: the recursive functions are then not refined for that property. They are still refined for a property that is unknown, or whose counterexample relies on outputs the contracts allow but the functions do not have, or cannot be checked in time; and the other callees are refined as before. A check such as `Fact(4) = 25` is then falsified in the first analysis of the caller, which is not analyzed again up to the limit.

When the check shows the counterexample spurious, the property cannot fail with the values the functions have, and the values of the calls it evaluated are not lost: those of the functions proved terminating (see below) are given to the refinements of the caller,

as facts on the functions, like the values of the calls at constant arguments. A check such as `x = 3 => Fact(x) = 6`, whose argument is an input that the check fixes, is then proved in the first refinement, rather than after as many refinements as `Fact(3)` needs unrollings, four, which the default limit of two does not allow. The option `--rec_learn_values false` turns this off.

In a modular analysis, compositional or not, a call at constant arguments to a recursive function that is proved terminating is evaluated instead: Kind 2 computes its value with the definition of the function, in a solver of its own, and gives it to the solver of the analysis, whether the contract abstracts the function or not. A function is proved terminating when its own analysis proved the termination checks of each of its recursive calls, or its measure is an ADT one, and so is every recursive function it calls. Only a modular analysis runs the analyses of the functions themselves, so in any other analysis the termination of a function is not known, and a call to it is left to its unrolling. In a compositional and modular analysis, `Fact(4) = 24` is then proved in the first analysis of the caller, although the contract of `Fact` does not give it, with no refinement; in a modular, non-compositional one, where `Fact` is unrolled, `Fact(12) = 479001600` is proved although it needs more unrollings than `--rec_unrollings` allows. An argument is constant when it is a literal, a local variable whose value is the same constant at every step, such as `b` in `a = 3; b = a + 1`, or the output of a call evaluated in turn, as the inner call of `Fact(Fact(3))`; an input, or an expression with a `pre` or a `->`, is not. A function with no contract is evaluated as well: it is unrolled rather than abstracted, and a call that needs more unrollings than `--rec_unrollings` allows, such as `Sum(20)` for a `Sum` that recurses down to 0, is then proved rather than left unknown. A call whose value cannot be computed in time, or is not unique, is left as it was.

A call to a recursive function may be applied to a quantified variable (see the limitations on quantifiers) when the recursive calls past its unrollings are left unconstrained, as above: such a call has no instance to be unrolled, so it is compiled to an application of the symbol of the function, and the function is defined at the SMT level, as a `define-funs-rec` block, so that the solver knows it at every argument the quantifier ranges over:

```
datatype Nat = Zero | Succ (p: Nat);

function rec Even (n: Nat) returns (r: bool)
con
  decreases n;
noc
```

```
let
  r = match n with | Zero : true | Succ(m) : not Even(m) end;
tel

node main () returns ();
let
  check forall (n: Nat) Even(Succ(n)) = not Even(n);
tel
```

The property is proved by unfolding the definition once. The definition also ties the other calls of the function to its body past their unrollings, so none of their counterexamples is spurious; the solver is the one unfolding the recursion then, and a query about the function for all its inputs may not terminate. The definition applies the recursive calls under their termination checks, so that it has a model whether or not the measure decreases; the checks remain properties.

A call applied to a quantified variable is accepted under the conditions of a call to a non-recursive function that can be inlined, whatever the analysis:

- The function has no contract with guarantees, modes or a refinement type on an output, or is declared `transparent`. A translucent function with a contract is abstracted by it in compositional analyses only, but calls to it are rejected in every analysis. Where a contract abstracts the function, its symbol would be constrained at the arguments of its instances only, an arbitrary function under the quantifier.
- Its body is made of equations that define all of its outputs and local variables, without assertions, and no output or local variable is of an array type. No definition is built from another body.
- Every function the body calls can be inlined, or is a recursive function that meets these conditions in turn. A function is defined together with the functions of its recursive group and the functions they call, and an imported function, in particular, is an arbitrary function under the quantifier.

Otherwise the call is rejected. For a polymorphic function, the rule applies to the instantiation that is called, with the type arguments of the call: `Id@<int>` may be applied to a quantified variable while `Id@<Nat>`, for a refinement type `Nat`, and `Id@<int^2>` may not, if `Id` returns a value of its type parameter. In the body of a polymorphic node, a type parameter of the node given as a type argument is taken to be an array type, whatever the instantiations of the node. Kind 2 warns when the definition is left out because the solver or logic does not take recursive definitions (Z3 and cvc5 do, under the inferred logic). The symbol is then an arbitrary function under

the quantifier, and a property that holds may be reported falsifiable: with `--smt_logic UFLIA`, for instance.

6.5.3 Benefits and limitations

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.

The downside of using functions in your model is that the IC3QE engine and the IC3IA engine with the Z3qe or cvc5qe options must shut down, since their current implementation cannot reason about the resulting system.

6.6 Lemmas

A *lemma* states a fact about its inputs in its contract and proves it in its body. It is declared like a function, with the `lemma` keyword and no `returns` clause:

```
function imported g (n: int) returns (m: int);
```

```
lemma g_increasing (n: int)
con
  guarantee g(n) > n;
noc

lemma g_twice (n: int)
con
  guarantee g(g(n)) > n;
noc
let
  g_increasing(n);
  g_increasing(g(n));
tel
```

Here `g_increasing` has no body: it is assumed rather than proved (see Lemmas without a body), while `g_twice` is proved from it.

A lemma must have a contract, and is always opaque: its callers only know what its contract says, never its body. Its body is subject to the same restrictions as the body of a function, and may hold equations of local variables, but its purpose is to invoke other lemmas, whose guarantees are what its own guarantees are proved from. Its local declarations are variables only (`var`): a lemma cannot declare local constants.

6.6.1 Invoking a lemma

A lemma is invoked in a *call statement*, a call written on its own, with no left-hand side, in the body of a node, a function or a lemma. This is the only place a lemma may be invoked (not in an expression, nor in a contract), and the only thing a call statement may invoke. At the invocation, the guarantees of the lemma are assumed for the arguments it is given, and each of its assumptions is a proof obligation of the caller, reported as a property named after the invocation and the assumption: for instance, `L[L21C3].assume[L10C3]` for the assumption at line 10 of a lemma `L` invoked at line 21. In a node, the invocation holds at every step; its arguments may be any expression, including one with `pre`:

```
node main (x: int) returns (ok: bool);
let
  g_twice(x);
  ok = g(g(x)) > x;
  check ok;
tel
```

An invocation inside an `if`, `when` or `match` block (see If statements and frame conditions) only takes effect in the branch it is written in. This is how a proof distinguishes cases. A branch that needs no invocation, because the guarantee follows directly there, contains the statement `auto;`, which does nothing and is only allowed in the body of a lemma:

```
datatype L = Nil | Cons (hd: int, tl: L);

function rec len (l: L) returns (n: int)
con
  decreases l;
noc
let
  n = match l with
    | Nil : 0
    | Cons (h, t) : 1 + len(t)
  end;
tel

lemma LenNonNeg (l: L)
con
  guarantee len(l) >= 0;
  decreases l;
noc
let
  match l with
  | Nil:
    auto;
  | Cons (h, t):
    LenNonNeg(t);
  end
tel
```

6.6.2 Proofs by induction

A lemma may invoke itself, directly or through a cycle of other lemmas, as `LenNonNeg` does. The guarantees of the recursive invocation are then the induction hypothesis, which is only justified if the recursion terminates: such a lemma needs a `decreases` clause, and its recursive invocations are checked to decrease exactly as the calls of a recursive function are (see Recursive functions). Unlike a function, a lemma needs no `rec` modifier. A lemma that does not invoke itself may omit the `decreases` clause.

6.6.3 How lemmas are analyzed

A lemma is analyzed like a function: its analysis proves its guarantees under its assumptions, from the guarantees of the lemmas it invokes, and checks the assumptions of those invocations. Which lemmas are analyzed follows the usual rules for choosing top-level nodes: by default, only the top-level nodes are, so a lemma that is invoked by another node is only assumed there, *not proved*. To prove every lemma along with the rest of the model, run a modular analysis (`--modular true`), or designate the lemma as the main node with `--lus_main`.

6.6.4 Lemmas without a body

A lemma may be declared without a body, ending with its contract. Its contract is then never proved: its guarantees are assumed wherever the lemma is invoked, and its assumptions, as for any lemma, are an obligation of the caller. A typical use is to state properties of an imported function, as `g_increasing` does above, or:

```
function imported len (l: L) returns (n: int);

lemma len_nil ()
con
  guarantee len(Nil) = 0;
noc

lemma len_cons (h: int; t: L)
con
  guarantee len(Cons(h, t)) = 1 + len(t);
noc
```

Such a lemma cannot invoke itself, so it takes no `decreases` clause. Nothing checks that its guarantees are true: a false one makes every analysis that invokes the lemma vacuous. When its guarantees only mention functions with a body, enabling the realizability checks (`--enable CONTRACTCK`) proves or refutes them. Note that a lemma with an empty body (`let tel`), or a body holding only `auto;`, is not a lemma without a body: its guarantees are proved from its assumptions alone.

6.7 Integer division and modulo

The integer division operator `div` and the modulo operator `mod` follow the Euclidean definition that SMT-LIB gives them: for a divisor `n` other than zero, `m div n` and `m mod n` are the unique integers `q` and `r` such that `m = n * q + r` and `0 <= r < |n|`. The remainder is thus never negative, and the quotient is rounded towards negative infinity when the divisor is positive and towards positive infinity when it is negative:

`m`	**`n`**	**`m div n`**	**`m mod n`**
`7`	`2`	`3`	`1`
`-7`	`2`	`-4`	`1`
`7`	`-2`	`-3`	`1`
`-7`	`-2`	`4`	`1`

This is not the definition used by C, where the quotient is rounded towards zero and the remainder takes the sign of the dividend (`-7 / 2` is `-3` and `-7 % 2` is `-1`), nor, by extension, the one used by the Lustre compilers that generate C code. The two definitions agree when both operands are non-negative, or when the divisor divides the dividend exactly, but they differ otherwise. For instance, `(-1) div 100` is `-1` in Kind 2 and `(-1) / 100` is `0` in C, so the property below is valid for Kind 2 but does not hold of the compiled code:

```
node div_example (i: int) returns (j: int);
let
  j = i div 100;
  check i < 0 => j < 0;
tel
```

A model whose verified properties must carry over to generated code should therefore either keep the operands of `div` and `mod` non-negative, or be compiled with an implementation of integer division that follows the Euclidean definition.

Kind 2 evaluates `div` and `mod` the same way wherever they appear, whether their operands are constants folded by the frontend or symbolic values handled by the SMT solver. A division by zero is not an error: as in SMT-LIB, `m div 0` and `m mod 0`

denote arbitrary integers, about which Kind 2 assumes nothing except that the same expression denotes the same value.

Division and modulo on machine integers follow the definition of C instead.

6.8 The `fby` operator

Kind 2 supports the binary *followed-by* operator of Lustre V6: `e1 fby e2` is `e1` at the first step and the previous value of `e2` afterwards, that is, it is syntactic sugar for `e1 -> pre e2`.

```
-- A two-deep shift register: 0, 0, a(0), a(1), ...
b = 0 fby 0 fby a;
-- A counter: 1, 2, 3, ...
n = 0 fby n + 1;
```

The operator is right-associative, so `0 fby 0 fby a` is `0 fby (0 fby a)`. It binds tighter than the binary arithmetic, comparison and Boolean operators, so `0 fby n + 1` is `(0 fby n) + 1`, and looser than `pre`, `not` and other prefix operators except unary minus, so `not c fby c` is `(not c) fby c` and `- a fby b` is `- (a fby b)`.

6.9 Conditional expressions

Kind 2 provides two forms of conditional expression. Both select between two branches based on a Boolean condition, but they differ in how the branches are evaluated.

The `if ... then ... else ...` expression has *eager* semantics:

```
x = if condition then expr1 else expr2;
```

The `when ... then ... else ...` expression has *lazy* semantics:

```
x = when condition then expr1 else expr2;
```

Both expressions evaluate to `expr1` when `condition` is true and to `expr2` otherwise. The difference is operational: with the eager `if` form, both branches are evaluated at every step regardless of the condition, whereas with the lazy `when` form

only the selected branch is evaluated; the branch that is not selected is not evaluated at all. The lazy form is useful when one branch is only meaningful (for instance, only satisfies the assumptions it relies on) when the condition selects it.

A branch of a `when ... then ... else ...` expression may contain temporal operators and node calls. Since the branch is evaluated only at the steps where it is selected, these are evaluated on the clock of the branch:

- `pre e` refers to the value of `e` the last time the branch was selected, which may be several steps earlier, and `e1 -> e2` is `e1` the first time the branch is selected.
- A node called in the branch is activated only at the steps where the branch is selected: its internal state does not advance at the other steps.

For example, in

```
x = when c then (0 -> pre x + 1) else 0;
```

`x` counts the steps at which `c` holds (starting from 0), whereas with `if c then (0 -> pre x + 1) else 0`, the previous value of `x` would be 0 whenever `c` was false at the previous step.

Each form has a corresponding statement-level block, described in the next section: `if` statements desugar to `if ... then ... else ...` expressions, while `when` and `cond` blocks desugar to `when ... then ... else ...` expressions.

6.10 Short-circuit Boolean operators

Besides the standard Boolean operators `and`, `or`, and `=>`, which evaluate both of their operands, Kind 2 provides short-circuit (lazy) variants that evaluate their right operand only when necessary:

- `e1 and then e2` (short-circuit conjunction): if `e1` is false, the result is false and `e2` is not evaluated.
- `e1 or else e2` (short-circuit disjunction): if `e1` is true, the result is true and `e2` is not evaluated.
- `e1 ==> e2` (short-circuit implication): if `e1` is false, the result is true and `e2` is not evaluated.

When the right operand *is* evaluated, these operators agree with their eager counterparts: `and then` with `and`, `or else` with `or`, and `==>` with the implication operator `=>`. The right operand is evaluated like a branch of a `when ... then ... else ...` expression: `e1 and then e2` behaves as `when e1 then e2 else false`, `e1 or else e2` as `when e1 then true else e2`, and `e1 ==> e2` as `when e1 then e2 else true`. In particular, temporal operators and node calls in the right operand are evaluated on the clock of the steps at which it is evaluated.

These operators are convenient when the right operand is only well-defined, or only satisfies its assumptions, when the left operand has the appropriate value, as in `x <> 0 and then y / x > 1`.

6.11 If statements and frame conditions

Within node definitions, Kind 2 has support for two features that allow the programmer to use a more imperative style– (1) `if` statements and (2) frame conditions.

6.11.1 If statements

In addition to the conditional *expressions* described above, in some circumstances it may be more natural to use `if` *statements* that serve as control flow (rather than evaluate to a value). For example, Kind 2 supports statements of the form:

```
if condition1 then
   y1 = expr1;
   y2 = expr2;
elsif condition2 then
   y1 = expr3;
   y2 = expr4;
else
   y1 = expr5;
   y2 = expr6;
fi
```

In the above block, if `condition1` is true, then `y1` and `y2` will be set to `expr1` and `expr2`, respectively. Otherwise, `y1` and `y2` will be set to either `expr3` and `expr4` or `expr5` and `expr6`, depending on the value of `condition2`. The `if` statement is closed with the `fi` token. As with other mainstream programming languages, Kind 2 allows for arbitrary nesting of `if` statements, as well as writing `if` statements that do not have any `else` or `elsif` blocks.

Note: If statements are syntactic sugar for conditional expressions. The `if` statement above is equivalent to:

```
y1 = if condition1 then expr1 else (if condition2 then expr3 else expr5);
y2 = if condition1 then expr2 else (if condition2 then expr4 else expr6);
```

Although this desugaring gives each assigned variable its own copy of the conditions, the conditions are evaluated only *once* per timestep: all variables assigned by the same block see the same condition values, and therefore take the same branch. The distinction matters when a condition contains a call to a node whose outputs are not uniquely determined by its inputs — for example, an imported node with an underspecified (or no) contract, or a node whose definition uses the `any` operator. Such a call is evaluated once per timestep for the whole block, and the resulting value is shared by all the equations generated from the block. In the following example, the property `"agree"` is invariant because `y1` and `y2` always take the same branch:

```
node imported nondet() returns (b: bool);

node example() returns (y1, y2: int);
let
   if nondet() then
      y1 = 1; y2 = 1;
   else
      y1 = 2; y2 = 2;
   fi
   check "agree" y1 = y2;
tel
```

The same rule applies to the guards of the `when` and `cond` blocks below, and to blocks appearing inside frame conditions (where a variable left undefined in a branch holds its previous value).

6.11.2 When blocks

Kind 2 also supports `when` blocks, which are similar in structure to `if` statements but use *lazy* branch semantics:

```
when condition1 then
   y1 = expr1;
   y2 = expr2;
```

```
else
   y1 = expr3;
   y2 = expr4;
end
```

Additional branches can be expressed by nesting `when` blocks inside the `else` branch:

```
when condition1 then
   y1 = expr1;
   y2 = expr2;
else
   when condition2 then
      y1 = expr3;
      y2 = expr4;
   else
      y1 = expr5;
      y2 = expr6;
   end
end
```

Semantics

At each step, only the selected branch is evaluated. In particular, branch expressions that are not selected are not evaluated. This is useful when one branch relies on assumptions that do not hold in other cases.

As for `if` blocks, `when` blocks are statement-level syntax sugar. For each assigned variable, the blocks above correspond to nested lazy `when ... then ... else ...` expressions. As with `if` blocks, each guard is evaluated at most once per timestep, and its value — including any nondeterministic choice made by a node call appearing in the guard — is shared by all the variables assigned by the block. Consistent with the lazy branch semantics, the guard of a nested `when` block is itself part of the enclosing branch: it is evaluated only when that branch is selected, and node calls appearing in it are activated on the enclosing guards (their internal state only advances at timesteps where the enclosing branch is selected).

Branch expressions may contain temporal operators and node calls, which are evaluated on the clock of the branch, as for the `when ... then ... else ...` expression.

The current restriction for `when` blocks is:

- `if` blocks cannot be nested inside `when` blocks, and `when` blocks cannot be nested inside `if` blocks.

6.11.3 Cond blocks

Kind 2 also supports `cond` blocks, which use a pattern-matching style with multiple guarded branches and an `otherwise` clause:

```
cond
  | condition1:
     y1 = expr1;
     y2 = expr2;
  | condition2:
     y1 = expr3;
     y2 = expr4;
  otherwise:
     y1 = expr5;
     y2 = expr6;
end
```

The semantics of `cond` blocks is the same as for `when` blocks: at each step, only the selected branch is evaluated, and branch expressions that are not selected are not evaluated.

As in `when` blocks, branch expressions may contain temporal operators and node calls. The current restriction for `cond` blocks is the same as for `when` blocks:

- `if` blocks cannot be nested inside `cond` blocks, and `cond` blocks cannot be nested inside `if` blocks.

6.11.4 Match blocks

A `match` block is to a match expression what an `if` block is to an if-then-else: the arms hold equations rather than a value.

```
datatype List = Nil | Cons (hd: int, tl: List);

node head_or (l: List; d: int) returns (y: int; found: bool);
let
  match l with
```

```
  | Cons (h, t):
    y = h;
    found = true;
  | Nil:
    y = d;
    found = false;
  end
tel
```

Each arm introduces the variables its pattern binds, in scope only within that arm. Patterns may be nested, and a bare identifier that is not a constructor matches anything.

The semantics is the same as for a match expression: at each step only the selected arm is evaluated, and the arms that are not selected are not.

Restrictions:

- The arms must be exhaustive, unless the block sits inside a frame block, where a value matched by no arm leaves the variables to stutter.
- Every variable defined in one arm must be defined in all of them, again unless the block sits inside a frame block.
- A pattern variable may not have the same name as an input, output or local of the enclosing node.
- Only equations, `when` blocks, nested `match` blocks and `auto` may appear in an arm.
- The scrutinee may not call a node or contain an `any` operator. Assign it to a local variable and match on that instead. Calls to functions, and the `choose` operator, are allowed.
- As with `cond` blocks, `match` blocks cannot be nested inside `if` blocks, and `if` blocks cannot be nested inside `match` blocks. `match` and `when` blocks may be nested inside each other in either direction.

6.11.5 Frame conditions

Kind 2 also has support for code blocks with frame conditions. At the beginning of the block (denoted by the `frame` keyword), the user specifies a list of variables that they wish to define within the frame block. All variables defined within the frame block must be present in this list. Then, initial values are optionally specified for these variables. Variables are defined within the frame block body (denoted by the `let` and `tel` keywords). It is possible to leave variables (partially or fully) undefined. A variable is

considered *fully undefined* if it is declared in the list of frame block variables, but there is no definition given in the frame block body. A variable is considered *partially defined* if it is defined within an `if` block, but a definition is not supplied in all cases (e.g., `if c then y = x; fi` defines `y` within the `then` case but not the `else` case).

If a variable is fully undefined, then it is set to its initialization value (if one exists) in the first timestep, and it stutters (is set equal to its value on the previous timestep) on other timesteps. If a variable is partially undefined, then the same assignment is performed, but only for branches of the `if` block where the variable is left undefined.

The following example involves three variables `y1`, `y2`, and `y3`. Since `y1` is left fully undefined within the frame block body, it will always be equal to 0 (its initialization value). `y2` will have value `0, 1, 2, 3, ...` since it is not fully or partially undefined (notice that the initialization value is not used in this case). Finally, `y3` will have value `//, 0, 1, 2, 3, ...` since it is also not fully or partially undefined, regardless of the presence of an unguarded `pre`.

```
node example() returns (y1, y2, y3: int);
let
   frame ( y1, y2, y3 )
   (* Initializations *)
   y1 = 0; y2 = 100; y3 = 5;

   (* Body *)
   let
      y2 = counter();
      y3 = pre counter();
   tel
tel


node counter() returns (y: int);
let
   y = 0 -> pre y + 1;
tel
```

Frame conditions are especially useful when combined with the `if` statements described in the previous subsection, as variables can be left undefined in some branches of the `if` statement.

```
node example() returns (y1, y2: int);
let
```

```
   frame ( y1, y2 )
   (* Initializations *)
   y1 = 10;
   y2 = 100;

   (* Body *)
   let
      if (counter() < 10)
      then
         y1 = counter();
      else
         y2 = counter() * 2;
      fi
   tel
tel


node counter() returns (y: int);
let
   y = 0 -> pre y + 1;
tel
```

In the above example, `y1` is left undefined in the `else` branch of the `if` statement, and `y2` is left undefined in the `then` branch. Since the condition `counter() < 10` holds in the initial timestep, `y1` will be initialized to `0` according to its definition in the `then` branch. Then, `y1` will continue to be equal to `counter()` on the second through tenth timesteps, and then stutter (staying at 9) for the remaining timesteps.

On the other hand, `y2` starts at its supplied initialization value at the top of the frame block (`100`) since it is left undefined in the `then` branch of the `if` block. It stutters there for the first 10 timesteps, and then is set to `counter() * 2` for the remaining timesteps.

Note that variables do not have to have initializations. When no initialization is given, a variable's initial value is equal to the initial value of the expression defined in the frame block body. If the corresponding expression is undefined in the first timestep, or if there is no equation defining the variable in the first timestep, then the variable is undefined in the first timestep. For example, the following code is supported because even though `y1` , `y2` , and `y3` do not have an initializations, they are present in the list of variables `frame ( y1, y2, y3 )` . The initial value of `y1` is 0 (the initial value assigned by `counter()`); the initial value of `y2` is undefined (due to the unguarded

`pre`); and the initial value of `y3` is also undefined (due to the lack of an equation defining `y3` initially).

```
frame ( y1, y2, y3 )
let
   y1 = counter();
   y2 = pre counter();
tel

node counter() returns (y: int);
let
   y = 0 -> pre y + 1;
tel
```

Also, it is still possible to assign to multiple variables at once (equations of the form `y1, y2 = (expr1, expr2);`) in either the initializations or the frame block body.

The frame block semantics may introduce unguarded `pre` expressions. For example, the definition of `y` in the following code block is equivalent to `y = pre y`. So, Kind 2 will produce two warning messages. The first will state that `y` is uninitialized in the frame block, and the second will state that there is an unguarded `pre` (due to this lack of initialization).

```
frame ( y )
let
tel
```

Similarly, in the following code block, the definitions of `y1` and `y2` are equivalent to `y1 = if cond then 0 else pre y1` and `y2 = if cond then pre y2 else 1`, respectively. This situation (and any other situation where the frame block semantics result in the generation of an unguarded `pre`) will also generate the two warnings as discussed in the previous paragraph.

```
frame (y1, y2)
let
   if cond
   then
      y1 = 0;
   else
      y2 = 1;
```

```
   fi
tel
```

6.11.6 The `last` operator

Within a frame block, the expression `last x` (where `x` is a local or output variable in scope) denotes the value of `x` at the *immediately preceding timestep*, with the value at the first timestep given by the frame's initialization of `x` (or left undefined when `x` has no initialization). `last x` always refers to the value of `x` one timestep earlier, regardless of any branch conditions under which it appears.

`last x` is **not** the same as writing `init_x -> pre x` inside a branch of a `when` (or `cond`) block. When `init_x -> pre x` is written directly inside a branch, the lazy branch semantics make `pre x` refer to the value of `x` the *last time that branch was selected*, which may be several timesteps earlier. In contrast, `last x` always refers to the value of `x` at the immediately preceding timestep. (In a context that is evaluated at every timestep — for example a plain frame-block equation outside any `when` / `cond` branch — the two coincide, but `last x` is the reliable way to express "value at the previous timestep" in all contexts.)

The `last` operator may only be used inside a frame block; using it elsewhere is an error.

For example, in the following frame block `last o` refers to the value of `o` at the previous timestep, initialized to `i` (the initialization of `o`):

```
frame (o)
o = i;
let
   o = last o + 1;
tel
```

The initialization of a frame block variable is evaluated only *once*, at the first timestep, and every reference to it denotes that same evaluation: the value a variable holds when the frame initialization applies and the value `last x` denotes at the first timestep are guaranteed to be the same. The distinction matters when the initialization expression is not uniquely determined — for example, an `any` operator or a call to an imported node. In the following example, the initial value of `x` is an arbitrary non-negative integer *chosen once*: at the first timestep, if `m` is false, `x` keeps that value,

and if `m` is true, `x` is that same value plus one — there are never two independent choices, one for `x` and another for `last x`. The property `"nonneg"` is invariant:

```
node count (m: bool) returns (x: int);
let
   frame (x)
   x = any { v: int | v >= 0 };
   let
      when m then
         x = last x + 1;
      end
   tel
   check "nonneg" x >= 0;
tel
```

The same guarantee applies element-wise to array initializations (for example, `x[i] = any { v: int | v >= 0 }` makes one shared choice per element).

6.11.7 Omitting equations in `when` blocks within frame blocks

Just as with `if` statements, a variable may be left undefined in some branches of a `when` block when the `when` block appears within a frame block. When the definition of a variable `x` is omitted in a branch, that branch behaves as if it contained the equation `x = last x` (i.e., `x` keeps its previous value, initialized by the frame). For example, the following two frame blocks are equivalent:

```
frame (o, c1, c2)
o = i; c1 = 0; c2 = 0;
let
   when m then
      o = last o + 1;
      c1 = 1 -> pre c1 + 1;
   else
      o = last o - 1;
      c2 = 1 -> pre c2 + 1;
   end
tel
```

```
frame (o, c1, c2)
o = i; c1 = 0; c2 = 0;
let
   when m then
```

```
      o = last o + 1;
      c1 = 1 -> pre c1 + 1;
      c2 = last c2;
   else
      o = last o - 1;
      c2 = 1 -> pre c2 + 1;
      c1 = last c1;
   end
tel
```

Within a frame block, the `else` branch of a `when` block (and the `otherwise` branch of a `cond` block) may also be omitted entirely. An omitted `else`/`otherwise` branch behaves as if it defined every frame block variable with `x = last x`. For example, the following two frame blocks are equivalent:

```
frame (o, c1, c2)
o = i; c1 = 0; c2 = 0;
let
   when m then
      o = last o + 1;
      c1 = 1 -> pre c1 + 1;
   end
tel
```

```
frame (o, c1, c2)
o = i; c1 = 0; c2 = 0;
let
   when m then
      o = last o + 1;
      c1 = 1 -> pre c1 + 1;
   else
      o = last o;
      c1 = last c1;
      c2 = last c2;
   end
tel
```

Outside a frame block, every variable defined in any branch of a `when` block must still be defined in all branches, and the `else`/`otherwise` branch cannot be omitted.

6.11.8 Restrictions

A frame block cannot be nested within an if statement or another frame block, as demonstrated in the following examples:

```
if condition
then
   frame ( y1, y2 )
   y1 = init1; y2 = init2;
   let
      y1 = 10;
   tel
fi
```

```
frame ( y1, y2 )
y1 = init1; y2 = init2;
let
   y1 = expr1;
   frame ( y2 )
   y2 = init3;
   let
      y2 = expr2;
   tel
tel
```

Assertions, `MAIN` annotations, and `PROPERTY` annotations also cannot be placed within if statements or frame blocks.

Since an initialization only defines a variable at the first timestep, it need not be stateful. Therefore, a frame block initialization cannot contain any `pre` or `->` operators. This restriction also ensures that initializations are never undefined.

6.12 Polymorphic nodes

In some situations, the user may want to express multiple variations of a node, where the only differences between them lie in the input and output types. For example, consider different interface type variations of the `SafePre` node, which returns the previous value of its single input, but initialized with the first value of the input stream.

```
node SafePreInt(x: int) returns (y: int);
let
  y = x -> pre x;
tel

node SafePreBool(x: bool) returns (y: bool);
let
  y = x -> pre x;
tel

node Top(x1: int; x2: bool) returns (y1: int; y2: bool);
let
  y1 = SafePreInt(y1);
  y2 = SafePreBool(y2);
tel
```

Kind 2 allows the user to express such variations more concisely through *polymorphic nodes*, where the user includes a set of polymorphic type parameters in the node declaration and the specific type arguments at the call site. Polymorphic type parameters are specified using angle brackets as `<ty1; ...; tyn>` whereas call-site polymorphic arguments are specified using the `@` instantiation operator.

```
node SafePre<T>(x: T) returns (y: T);
let
  y = x -> pre x;
tel

node Top(x1: int; x2: bool) returns (y1: int; y2: bool);
let
  y1 = SafePre@<int>(y1);
  y2 = SafePre@<bool>(y2);
tel
```

Note that `SafePre` can be called with any type, not just primitive types (e.g. `SafePre@<[int, bool]>(.)` and `SafePre@<[int, U]>(.)`, where `U` is itself a type parameter in the caller's declaration).

Kind 2 can also infer the type arguments of a polymorphic call, so the `@<...>` instantiation is optional in most cases. When no type arguments are supplied, Kind 2 determines them *bottom-up* by unifying the node's input parameter types against the

types of the actual arguments at the call site. For instance, the two calls in `Top` above can be written without any annotation:

```
node Top(x1: int; x2: bool) returns (y1: int; y2: bool);
let
  y1 = SafePre(y1);
  y2 = SafePre(y2);
tel
```

Here `T` is inferred to be `int` in the first call and `bool` in the second.

Inference uses *base types only*: any refinement, subrange, or history information on the arguments is stripped before unification, so the inferred type argument is always the underlying base type. For example, if an argument has type `subrange [0, 10] of int` (or a refinement type over `int`), the corresponding type parameter is inferred as `int`.

A type argument can be inferred only when the corresponding type parameter appears in an *input* position, so that it can be determined from the arguments. If a type parameter occurs only in the node's outputs (and thus cannot be recovered from the call arguments), the `@<...>` annotation must still be provided explicitly; otherwise Kind 2 reports that the call requires an explicit annotation. When an explicit annotation is given, it must be consistent with the types of the arguments, or a type error is raised.

Another example is a polymorphic node `PairSwap`, which takes a polymorphic pair tuple as input and returns the corresponding swapped pair tuple as output.

```
node PairSwap<T; U>(x: [T, U]) returns (y: [U, T]);
let
y = {x[1], x[0]};
tel
```

For a polymorphic node to be well-typed, it must be meaningful for *any* type instantiation (in other words, the type parameters are semantically universally quantified). This type of polymorphism is called *parametric polymorphism*, and is also sometimes referred to as *generics* in general-purpose programming languages.

To illustrate these semantics, even though the `+` operator is overloaded between `int -> int -> int` and `real -> real -> real`, the following polymorphic node will give a type error, as it cannot be instantiated with any type.

```
-- Generates a type error
node BadPolymorphicAdd<T>(x1, x2: T) returns (y: T);
let
  y = x1 + x2;
tel
```

Note that polymorphic nodes can have `check(.)` statements just as non-polymorphic nodes. When checking properties of polymorphic nodes at the top level, the type parameters are interpreted as abstract types.

A recursive function can be polymorphic too. Kind 2 analyzes a copy of a polymorphic node for each list of type arguments it is called with, so a call within a recursive group (that is, a call to the function itself, or to a function it is mutually recursive with) cannot instantiate a type parameter with a type built from a type parameter of the caller, which would require infinitely many copies. Each type argument of such a call must be either one of the caller's type parameters, or a type that mentions none of them:

```
datatype List<T> = Nil | Cons (hd: T, tl: List<T>);

function rec Length<T>(l: List<T>) returns (n: int);
(*@contract
  decreases l;
*)
let
  n = match l with | Nil : 0 | Cons (_, tl) : 1 + Length@<T>(tl) end;  -- Accepted
tel

function rec F<T>(n: int; x: T) returns (r: int);
(*@contract
  decreases n;
*)
let
  r = when n <= 0 then 0 else F@<List<T>>(n - 1, Cons(x, Nil@<T>));  -- Rejected
tel
```

6.13 Polymorphic contracts

In addition to polymorphic nodes, Kind 2 supports polymorphic contracts. The first way of defining a polymorphic contract is by adding a type parameter to a contract

definition. For example, the `Stutter` contract states that the output `y` must either be equal to the input `x` or the previous value of `x`.

```
contract Stutter<T> (x: T) returns (y: T) ;
let
   guarantee
     (y = x) or
     (true -> (y = pre x));
tel
```

Then, the polymorphic contract can be included in a node using an import statement, where the type arguments are provided at the import statement (analogously to a polymorphic node declaration and node call).

```
contract Stutter<T> (x: T) returns (y: T) ;
let
   guarantee
     (y = x) or
     (true -> (y = pre x));
tel

node N (x: int) returns (y: int);
con
   import Stutter@<int>(x) returns (y);
noc
let
   y = pre x;
tel


node P<U>(x: U) returns (y: U);
con
   import Stutter@<U>(x) returns (y);
noc
let
   y = pre x;
tel
```

Above, node `N` instantiates the contract `Stutter` with type `int`. Also, node `P` demonstrates using a polymorphic contract declaration with a polymorphic node.

Another way of specifying a polymorphic contract is by including it directly in the node declaration of a polymorphic node as a local contract.

```
node M<T>(x: int) returns (y: int);
con
   guarantee
      (y = x) or
      (true -> (y = pre x));
noc
let
   y = pre x;
tel
```

6.14 Nondeterministic choice operator

There are situations in the design of reactive systems where nondeterministic behaviors must be modeled. Kind 2 offers a convenient polymorphic operator of the form `any { x: T | P(x) }` which denotes an arbitrary stream of values of type `T` satisfying the predicate `P`. We also support `choose { x: T | P(x) }`, where the only difference between `any` and `choose` is that `any` is nondeterministic, while `choose` is functional (deterministic and non-temporal). In the expression above `x` is a locally bound variable of Lustre type `T`, and `P(x)` is a Lustre boolean expression that typically, but not necessarily, contains `x`. The expression `P(x)` may also contain any input, output, or local variable that are in the scope of the `any` (or `choose`) expression. The following example shows a component using the `any` (or `choose`) operator to define a local stream `l` of arbitrary odd values.

```
node N(y: int) returns (z:int);
con
  assume "y is odd" y mod 2 = 1;
  guarantee "z is even" z mod 2 = 0;
noc
  var l: int;
let
  l = any { x: int | x mod 2 = 1 };
  -- with `choose`, `l` is constant
  -- l = choose { x: int | x mod 2 = 1 };
  z = y + l;
tel
```

In reality, the polymorphic operator `any` (or `choose`) can be instantiated with any Lustre type `T` using the instantiation operator `@` as follows: `any@<T>`. For instance, the expression `any@<int>` is also accepted and denotes an arbitrary stream of

values of type `int`. In fact, the form `any { x: T | P(x) }` is syntactic sugar for the more verbose form `any @ < subtype { x: T | P(x) } >`, where `T` has been instantitated with the refinement type `subtype { x: T | P(x) }`.

A challenge for the user with the use of the `any` (or `choose`) operator arises if the specified condition is inconsistent, or more generally, unrealizable. In that case, the system model may be satisfied by no execution trace. As a consequence, any property, even an inconsistent one, would be trivially satisfied by the (inconsistent) system model. For instance, the condition of the `any` (or analogously, `choose`) operator in the node of the following example is inconsistent, and thus, there is no realization of the system model. As a result, Kind 2 proves the property P1 valid.

```
node N(y: int) returns (z: int);
  var l: int;
let
  l = any { x : int | x < 0 and x > 0 };
  -- Use `choose` if you want `l` to be constant
  -- l = choose { x : int | x < 0 and x > 0 };
  z = y + l;
  check "P1" z > 0 and z < 0;
tel
```

This problem is mitigated by the possibility for the user to check that the predicate `P(x)` in the `any` (or `choose`) expression is realizable. This is possible because, for each `any` (resp., `choose`) expression occurring in a model, Kind 2 introduces an internal imported node (resp., imported function) whose contract restricts the values of the returned output using the given predicate as a guarantee. The user can take advantage of this fact to detect issues with the conditions of `any` (or `choose`) expressions by enabling Kind 2's functionality that checks the realizability of contracts of imported nodes and functions. When this functionality is enabled, Kind 2 is able to detect the problem illustrated in the example above.

It is worth mentioning that Kind 2 does not consider the surrounding context when checking the realizability of the introduced imported node or function. Because of this limitation, some checks may fail even if, in a broader context where all constraints included in the model are considered, the imported node or function would actually be considered realizable. Only the constraints imposed by the variable types are taken into account.

For instance, the realizability check for the `any` expression in the following example would fail if `b` were declared simply as an integer stream, rather than using the refinement type `subtype { x: int | a <= x }`.

```
node N(a: int) returns (z: int);
var b: subtype { x: int | a <= x };
let
  b = a + 10;
  z = any { x: int | a <= x and x <= b };
  check z>=a+10 => z=b;
tel
```

7 Arrays

> **Experimental feature**

7.1 Lustre arrays

Kind 2 supports the traditional Lustre V5 syntax for arrays.

7.1.1 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 Extension to unbounded arrays).

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 = struct { 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.

7.1.2 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]`.

7.1.3 Element update

A new array that is identical to an existing one except at selected indices can be constructed with the *element update* syntax `A[i := v]`. It denotes a copy of the array `A` in which the element at index `i` is replaced by `v`; the original array `A` is not modified. For example,

```
B = A[0 := 1];
```

defines `B` to be equal to `A` everywhere except at index `0`, where its value is `1`. Several elements can be updated at once by separating the individual updates with semicolons:

```
B = A[0 := 1; 2 := 3];
```

The update syntax can be combined with selection to update an element of a multidimensional array or of an array of compound values. For instance, given the matrix `M` above, `M[1 := M[1][2 := 0]]` denotes a copy of `M` in which the element at row `1`, column `2` is set to `0`.

7.1.4 Array comprehensions

An array can also be defined element by element with an *array comprehension*. Where the expression `v ^ n` is the array of size `n` whose elements are all `v`, the expression

```
(e foreach i) ^ n
```

is the array of size `n` whose element at index `i` is the value of `e`, for every `i` between `0` and `n - 1`. The index variable `i` is an integer bound in `e`, where it shadows any variable or constant of the same name, and the size `n` must be a constant expression, as in the type `t ^ n`. For example,

```
C = (A[i] or B[i] foreach i) ^ n;
```

defines `C` as the element-wise disjunction of the Boolean arrays `A` and `B`, and `(i * i foreach i) ^ 4` is the array `[0, 1, 4, 9]`.

A comprehension may list several index variables, one for each dimension of a multidimensional array, and is then followed by one size per index. The sizes are written as in the type of the array, so they are given from the innermost dimension to the outermost one, while the index variables are listed from the outermost to the innermost: if `A` and `B` have type `bool ^ n ^ m`, an array of `m` arrays of size `n`,

```
C = (A[i][j] or B[i][j] foreach i, j) ^ n ^ m;
```

has the same type, with `i` ranging over the `m` rows and `j` over the `n` columns. A caret after the last size replicates the array, as for any array: `(i foreach i) ^ 3 ^ 2` is `[[0, 1, 2], [0, 1, 2]]`.

The sizes can also be written next to the index variables they bound, which keeps each size with its index: `(e foreach i < m, j < n)` is `(e foreach i, j) ^ n ^ m`. For instance, the array `C` above is also

```
C = (A[i][j] or B[i][j] foreach i < m, j < n);
```

and `(i * i foreach i < 4)` is the array `[0, 1, 4, 9]`. A comprehension gives the sizes of all of its index variables in the same way, either bounding them or after it. A bound extends to the next `,` or `)`, and, as the sizes after a comprehension, it is a constant outside the scope of the index variables: `(e foreach i < m, j < i)` is rejected, rather than defining a triangular array. A caret after a comprehension whose index variables are bounded replicates the array: `(i foreach i < 3) ^ 2` is `[[0, 1, 2], [0, 1, 2]]`. As its sizes are within the parentheses, an element of such a comprehension is selected directly: `(i * i foreach i < 4)[2]` is `4`.

The body of a comprehension may itself be an array, such as a comprehension: `((i + j foreach j) ^ 3 foreach i) ^ 2` is `[[0, 1, 2], [1, 2, 3]]`, and the inner body may refer to the outer index variables.

The body may contain temporal operators. In particular, an element may depend on the previous value of the array being defined, as in this sliding window over a stream `s`, which stores the last `n` values of `s` (with `-1` for the values before the first step).

```
A = (if i = 0 then s else (-1 -> pre A[i-1]) foreach i) ^ n;
```

A temporal operator in a branch of a `when` expression, in an arm of a `match` expression, or in the right operand of `and then`, `or else` or `==>`, is evaluated only when its condition holds, which Kind 2 compiles as a call to a node. If the condition mentions an index variable, this is supported only when the size of the comprehension is a constant, as in `(when i = 0 then s else (-1 -> pre A[i-1]) foreach i) ^ 3`. With a symbolic size, such as a constant input of the node, it is rejected, as is a call to a node in the same position. The eager conditional above can be used instead.

As for `v ^ n`, a selection binds more tightly than the sizes, so a comprehension is written in parentheses before selecting one of its elements: `((i * i foreach i) ^ 4)[2]` is `4`.

A comprehension is a value, not a definition: its body cannot refer to the current value of the array being defined, which would be a cyclic definition. Comprehensions are

supported in the body of nodes and functions, including assertions and properties, and in contracts, including ghost variables, assumptions, guarantees and modes. They are not supported in `decreases` clauses, in the ghost constants of a contract and in types (such as the predicate of a refinement type), and they cannot mention a variable bound by an enclosing quantifier or match arm.

A global or local constant may also be defined by a comprehension, as in

```
const N: int;
const Sq: int^N = (i * i foreach i) ^ N;
```

Its definition is then used in place of the constant wherever the constant appears, in node bodies and in contracts, so the constant cannot be used in a type, such as the size of an array.

7.1.5 Structural equality

Arrays support **structural equality** (denoted by `=`) and **structural disequality** (denoted by `<>`). Two arrays are structurally equal when they have the same size and their elements are equal index by index. For example, given the array `A` defined by `A = [2, 5, 7]`, the expression `A = [2, 5, 7]` is valid, while `A <> [2, 5, 8]` is valid as well. Structural equality also applies to multidimensional arrays and to arrays whose elements are themselves compound values (records, tuples, sets, maps, etc.).

7.1.6 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

7.2 Extension to unbounded arrays

Kind 2 extends Lustre to arrays whose size is an *unbounded* constant, that is, a constant whose value is not given. Array comprehensions define such arrays element-wise, and quantifiers in specifications express properties of all their elements. This allows one to model *parameterized* systems: contrary to the Lustre compilers, Kind 2 does not require the constants used as array sizes to be instantiated with actual values, and checks the properties *for any* array sizes.

7.2.1 Element-wise definitions

Earlier versions of Kind 2 defined arrays element-wise with equations of the form `A[i] = <term(i)>;`. These equations are no longer supported: Kind 2 rejects them, and an array comprehension or a recursive function defines the array instead.

- When `<term(i)>` does not refer to `A`, or only to previous values of `A` (under a `pre`), the equation becomes `A = (<term(i)> foreach i) ^ n;`, where `n` is the size of `A`. An array with several dimensions takes one index variable per dimension: `M = (if i = j then 1 else 0 foreach i, j) ^ n ^ n;` defines the identity matrix of size `n`.

- When an element is defined from other elements of the current value of `A`, as in `A[i] = if i = 0 then 0 else A[i-1] + 1;`, the array cannot be defined by a comprehension, whose body cannot refer to the current value of the array being defined. A recursive function computes such elements instead, as in the sum below.

- When an element is defined both from previous values of `A` and from other elements of its current value, the recursive function takes the previous value of `A` as an argument, since a function cannot use `pre` itself. For instance, `A[i] = 0 -> (if i = 0 then pre A[i] + 1 else A[i-1]);` becomes

```
function rec next (p: int^3; k: int) returns (v: int)
(*@contract
  decreases k;
*)
let
  v = when k <= 0 then p[0] + 1 else next(p, k - 1);
tel
```

and `A = (0 -> next(pre A, i) foreach i) ^ 3;`.

- A comprehension cannot mention a variable bound by a match arm. An array defined in the arms of a match block from such variables is defined outside the block instead, from a local variable that the arms define:

```
match t with
| A (v): x = v;
| B (w): x = w;
end
a = (x + i foreach i) ^ 3;
```

Examples

Sum of the elements in an array

The following node returns the sum of all elements in an array, computed by a recursive function that adds the first `k` elements of the array.

```
function rec sum_upto (const n: int; A: int^n; k: int) returns (s: int)
(*@contract
  decreases k;
*)
let
  s = when k <= 0 or k > n then 0 else sum_upto(n, A, k - 1) + A[k - 1];
tel

node sum (const n: int; A: int^n) returns (s: int);
let
  s = sum_upto(n, A, n);
tel
```

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

Array slices

```
node slice (const n: int; A: int ^ n; const low: int; const up: int)
returns (B : int ^ (up-low));
let
```

```
  B = (A[low + i] foreach i) ^ (up - low);
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 = (A[i] or B[i] foreach i) ^ n;
tel
```

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

Parameterized systems

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

  --%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`.

7.2.2 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.

Concise refinement type syntax

A quantified variable can also be given a refinement type using the concise syntax `x: type | Q(x)`, which restricts the quantification to the values of `x` of the given `type` that satisfy the predicate `Q(x)`. For example,

```
forall (i: int | 0 <= i and i < n) ok[i]
```

is equivalent to

```
forall (i: int) 0 <= i and i < n ==> ok[i]
```

For existential quantification the predicate is conjoined instead, so
`exists (i: int | Q(i)) P(i)` is equivalent to
`exists (i: int) Q(i) and then P(i)`.

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 = (if i = 0 then s else (-1 -> pre A[i-1]) foreach i) ^ n;
  ok = (A[i] = -1 or A[i] = s or (false -> pre ok[i]) foreach i) ^ n;

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

7.2.3 Limitations

One major limitation that is present in the arrays of Kind 2 is that one cannot have node calls in array comprehensions of unbounded size whose arguments contain the index variables of the comprehension.

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

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

A, B: int^n;
...

A = (some_node(B[i]) foreach i) ^ n;
```

Another limitation is that quantified variables cannot appear in the parameters of a node call. These limitations do not apply if the call is to an *inlinable* function, which is currently defined as a function that meets all the following criteria:

- It has a single output, and the output is defined by an equation.
- Either there is no proof obligation on its output (via a contract or a refinement type), or the function is annotated as transparent.
- It does not include `assert` statements, and none of its outputs and local variables has an array type. An array comprehension in its body counts as a local variable of an array type.

A quantified variable may also appear in the arguments of a call to a recursive function that Kind 2 defines at the SMT level. The function must meet the second and third criteria above, whatever the analysis: there is no proof obligation on its outputs, or it is annotated as transparent, and it has no `assert` statement and no output or local

variable of an array type. So must the functions of its recursive group and the recursive functions it calls, and any other function it calls must be inlinable. Such a call is compiled to an application of the symbol the function is defined as rather than to an instance of the function, so the quantifier is free to range over its arguments. This does not extend to a symbolic array index, which still cannot appear in the arguments of a call that is not inlined.

7.2.4 Command line options

We provide different encodings of array comprehensions 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 recursive functions for arrays (for cvc5)

The default encoding will use quantified formulas for array comprehensions.

For example if we have

```
A : int^6;
...
A = (x foreach k) ^ 6;
```

we will generate internally the constraint

> *∀ k: int. 0 <= k < 6 => (select A k) = x*

These form of constraint are handled in an efficient way by cvc5 (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 cvc5 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 = (x foreach k) ^ 6;
```

will now be encoded by the constraint

> *(select A 0) = x ⋀ (select A 1) = x ⋀ (select A 2) = x ⋀ (select A 3) = x ⋀ (select A 4) = x ⋀ (select A 5) = x*

`--arrays_rec`

This uses a special kind of encoding to tell cvc5 to treat quantified definitions of some uninterpreted functions as recursive definitions.

8 Machine Integers

Kind2 supports both signed and unsigned versions of C-style machine integers.

8.1 Declarations

Machine integer variables can be declared as global, local, or as input/ouput of nodes. Signed machine integers are declared as type `sint<N>` and unsigned machine integers are declared as type `uint<N>` where `N` is the width (some concrete positive integer).

The following

```
x : uint<8>;
y : sint<17>;
```

declares a variable `x` of type unsigned machine integer of width 8, and variable `y` of type signed machine integer of width 17.

8.2 Values

Machine integers values can be constructed using implicit conversion functions applied to integer literals. The implicit conversion functions are of the form `uint@<N>` for unsigned machine integers and `sint@<N>` for signed machine integers.

The following

```
x = uint@<8> 27;
y = sint@<17> -5012;
```

defines `x` to have value `27`, and `y` to have value `-5012`, given that `x` is a variable of type `uint<8>` and `y` is a variable of type `sint@<17>`.

8.3 Semantics

Machine integers of width `x` represent binary numbers of size `x`. Signed machine integers are represented using 2's complement.

The bounds of selected machine integers are specified here for convenience:

```
uint<8>  : 0 to 255
uint<16> : 0 to 65535
uint<32> : 0 to 4294967295
uint<64> : 0 to 18446744073709551615
sint<8>   : -128 to 127
sint<16>  : -32768 to 32767
sint<32>  : -2147483648 to 2147483647
sint<64>  : -9223372036854775808 to 9223372036854775807
```

When the conversion functions are used for literals that are out of this range, they are converted to a machine integer that is in range using the modulo operation, as in C. For instance, in the following

```
x = uint<8> 256;
y = sint<16> 32768;
```

`x` evaluates to `0` and `y` to `-3268`, given that `x` is a variable of type `uint<8>` and `y` is a variable of type `sint<16>`.

Conversions are allowed between machine integers of different widths, as long as both types are either signed or unsigned. Values remain unchanged when converted from a smaller to a larger width; values are adjusted modulo the range of the destination type when converted from larger to smaller width. The following code illustrates this.

```
a : sint<8>;
b : sint<16>;
c : uint<16>;
d : uint<8>;
a = sint<8> 120;
b = sint<16> a; -- b == sint<16> 120
```

```
c = uint<16> 300;
d = uint<8> c; -- c == uint<8> 44
```

8.4 Operations

Kind2 allows the following operations over the machine integer types.

8.4.1 Arithmetic Operations

Addition (`+`), subtraction (`-`), multiplication (`*`), division (`div`), modulo (`mod`), and unary negation (`-`) are allowed on either signed or unsigned machine integers, and return a machine integer with the same sign and same width as the input(s).

```
a, a1, a2 : uint<8>;
b : uint<16>;
c : uint<32>;
d : uint<64>;
e, f : sint<8>;
a1 = (uint@<8> 5);
a2 = (uint@<8> 22);
a = a1 + a2;
b = (uint@<16> 20) * (uint@<16> 200);
c = (uint@<32> 500) div (uint@<32> 5);
d = (uint@<64> 25) mod (uint@<64> 10);
e = (sint@<8> -5) + (- (sint@<8> 10));
f = (sint@<8> 10) - (sint@<8> -5);
```

Division and modulo on signed machine integers round the quotient towards zero and give the remainder the sign of the dividend, as C does: `(sint@<8> -7) div (sint@<8> 2)` is `-3` and `(sint@<8> -7) mod (sint@<8> 2)` is `-1`. Note that this differs from the Euclidean definition used by `div` and `mod` on the unbounded `int` type, where the same quotient is `-4` and the same remainder is `1`.

8.4.2 Logical Operations

Conjunction (`&&`), disjunction (`||`), and negation (`!`) are performed in a bitwise fashion over the binary equivalent of their machine integer inputs. Conjunction and disjunction are binary, while negation is unary. All 3 operations return a machine integer that has the same sign and same width as its input(s).

```
a, b, b1, b2, c : uint<8>;
a = (uint@<8> 0) && (uint@<8> 45); --a = (uint@<8> 0)
b1 = (uint@<8> 255);
b2 = (uint@<8> 45);
b = b1 && b2; --b = (uint@<8> 45)
c = !(uint@<8> 0); --c = (uint@<8> 255)
```

8.4.3 Shift Operations

Left shift (`lsh`) and right shift (`rsh`) operations are binary operations: the first input is either signed or unsigned, the second input is unsigned, and the sign of the output matches that of the first input; both inputs and the output have the same width.

Right shifting when the first operand is signed, results in an arithmetic right shift, where the bit shifted in matches the sign bit.

A left shift is equivalent to multiplication by 2, and a right shift is equivalent to division by 2, as long as the result can fit into the machine integer of the same width. In other words, the left shift operator shifts towards the most-significant bit and the right shift operator shifts towards the least-significant bit.

```
a, b, c : bool;
a = (uint@<8> 0) lsh (uint@<8> 10) = (uint@<8> 0); --true
b = (uint@<8> 255) rsh (uint@<8> 12) = (uint@<8> 255); --true
c = (sint@<8> -1) lsh (uint@<8> 1) = (sint@<8> -2); --true
```

8.4.4 Comparison Operations

The following comparison operations are all binary: `>` , `<` , `>=` , `<=` , `=` . They input machine integers of the same size and sign, and output a boolean value.

```
a : bool;
a = (sint@<8> -12) < (sint@<8> 12); --true
```

8.4.5 Cast Operations

In addition to casts to signed and unsigned integers, signed and unsigned integers can also be casted to mathematical integers with the `int` cast operator.

```
a : sint<8>;
b : uint<8>;
c : int;
d : int;
c = int a;
d = int b;
```

8.4.6 Concise Syntax

For signed and unsigned machine integers of widths 8, 16, 32, and 64, we support the concise syntax of `intN` for signed integers and `uintN` for unsigned integers (where `N` is 8, 16, 32, or 64). This syntax works for both types and cast operators.

```
a : int8;
b : uint8;
a = int8 0;
b = uint8 0;
```

8.5 Limitations

Currently, only SMT solvers cvc5 and Z3 support logics that allows the usage of integers and machine integers together. To use any of the other supported SMT solvers, the Lustre input must contain only machine integers.

Moreover, the IC3QE engine requires either cvc5 or Z3, and the IC3IA engine requires MathSAT, cvc5, or Z3, to run on models with machine integers. If these requirements are not satisfied, Kind 2 runs with the corresponding IC3 model checking engine disabled.

9 Refinement Types

Kind 2 supports refinement types. A refinement type is comprised of two components: (i) a **base type**, and (ii) a predicate that restricts the members of the base type.

9.1 Declarations

Refinement types have syntax of the form `subtype { var: base_type | P(var) }`.

For example, `type Nat = subtype { x: int | x >= 0 }` declares a refinement type `Nat` over the base type `int`, where the values of `Nat` are all the nonnegative integers. When assigning a refinement type to a node input, output, or local variable, Kind 2 also supports an alternative, more concise syntax of the form `var: base_type | P(var)`.

For example,

```
node N(x: int | x >= 0) returns (y: int | y >= 0);
```

denotes the interface of a node `N` which takes a stream of natural numbers `x` as input and returns a stream of natural numbers `y` as output. The above example can be equivalently expressed using the full syntax:

```
node N(x: subtype { n: int | n >= 0 }) returns (y: subtype { n: int | y >= 0});
```

The base type being refined can be *any* type, not just a primitive type. For example,

```
type Nat = subtype { x: int | x >= 0 };
type LessThan100 = subtype { x: Nat | x < 100 };
```

declares a refinement type `LessThan100` whose base type `Nat` is itself a refinement type. Note that we can still recursively chase base types until we reach a primitive type. In this case, `LessThan100`'s recursively chased primitive base type is `int`.

Additionally, refinement types can be components of more complicated types:

```
const n: int;
type Nat = subtype { x: int | x >= 0 };
type NatArray = Nat^n;
```

Above, we declare a type `NatArray`, an array of natural numbers.

Since Lustre is a declarative language, there is no conceptual ordering between variable declarations (input, output, and local variables). A consequence of this is that refinement type predicates can contain variables that are defined before *or after* the current variable in the input file. For example, the following is legal.

```
node N() returns (x: | x <= y; y | y = x + 10);
```

Above, the predicate in `x`'s type references `y`, which is allowed even though `y` comes after `x` in the list of node outputs.

9.2 Semantics

Refinement types on input variables represent assumptions, while refinement types on locals and node outputs represent proof obligations.

Consider the following example:

```
type Even = subtype { n: int | n mod 2 = 0 };
type Odd = subtype { n: int | n mod 2 = 1 };

node M(x1: Even; x2: Odd) returns (y: Odd);
let
   y = x1 + x2;
   --%MAIN;
tel
```

Kind2 will attempt to prove that node `M`'s output `y` respects type `Odd` while assuming that input `x1` has type `Even` and input `x2` has type `Odd`. More intuitively, Kind 2 will prove that adding an even and an odd integer results in an odd integer. Conceptually, the refinement types can be viewed as an augmentation of `M`'s contract as follows:

```
node M(x1: int; x2: int) returns (y: int);
con
   assume x1 mod 2 = 0;
   assume x2 mod 2 = 1;
   guarantee y mod 2 = 1;
noc
let
   y = x1 + x2;
   --%MAIN;
tel
```

9.3 Operations

From the point of view of primitive operations (e.g. `+`, `-`, `pre`) and node calls, variables with refinement types can syntactically be used anywhere that variables with the corresponding base type can be used, and vice versa. For example, if `x` has type `Nat`, `y` has type `Nat`, and `z` has type `int`, then `x+y`, `z+x`, and `y+z` (among other combinations) are all legal. Further, if node `M` has a single parameter of type `Nat`, then the node call `M(z)` is legal, and if node `N` has a single parameter of type `int`, then the node call `M(x)` is legal.

While all of the above are syntactically valid, Kind 2 may still fail type-related proof obligations. For example, in the node call `M(z)` (where `z` has type `int` and `M` takes a single parameter of type `Nat`), `M`'s typing assumption on its input will be violated if `z` can be negative.

9.4 Type Ascription

To check if an expression satisfies a refinement (or subrange) type, one can use a type ascription operator of the form `(e: T)`. The type ascription operator generates a proof obligation that `e` satisfies type `T`; it does *not* introduce an assumption that `e` satisfies type `T`. For example, the ascription `(1: Nat)` would introduce a proof obligation that `1` is a natural number (assuming `Nat` is a type capturing the natural numbers); this proof obligation would be discharged by Kind 2. Assuming `x` is an input variable of type `int`, the `check` statement `check (x: Nat) >= 0` would generate two proof obligations: First, it would generate the proof obligation associated with the `check` statement that `x >= 0`, and second, it would generated the proof obligation that `x` satisfies type `Nat`. Both these proof obligations would fail because

Kind 2 cannot prove that `x` is a natural number, as it is an input of type `int`. Again, the ascription introduces a proof obligation, not an assumption.

Ascriptions can also be used with non-refinement types. For example, the ascription `(1 + 2: bool)` would trigger a type checking error by Kind 2 before reaching the model checking phase. On the other hand, `(false or true: bool)` would pass type checking but not generate any proof obligations.

9.5 Realizability

Because refinement types are essentially contract augmentations, it is possible to specify refinement types that are *unrealizable*. In other words, it is possible to specify refinement type contraints that are unimplementable (impossible to satisfy with any implementation).

As an example, the following node interface is unrealizable:

```
node M(x: int) returns (y: int | 0 <= y and y <= x);
```

Output variable `y`'s refinement type states that `y` must be between `0` and `x`. However, if input `x` is negative, then no value for `y` will satisfy its type.

One way to make the above interface realizable is to add a refinement type for `x`:

```
node M(x: int | x >= 0) returns (y: int | 0 <= y and y <= x);
```

To check the realizability of refinement types, one can call `kind2 <filename> --enable CONTRACTCK`. Kind 2 performs three types of realizability checks:

1. Node and imported node contracts, including type information
2. Node environments, i.e., checking that the set of assumptions on a node's input is realizable
3. Individual refinement types, i.e., that a global refinement type declaration is realizable

You can specify a particular node or function to analyze using `--lus_main <node_name>`, a specific refinement type using `--lus_main_type <type_name>`, or a specific constant using `--lus_main_const <const_name>`.

9.6 Constants

Refinement types can also be assigned to constants, and Kind 2 treats defined and free constants differently. Both cases are illustrated with the refinement type:

```
type Pos = subtype { x: int | x > 0 };
```

A *defined* constant — one given a value — produces a **proof obligation** that the value satisfies the refinement predicate:

```
const c: Pos = 5;   -- Kind 2 checks that 5 > 0
```

A *free* constant — one declared without a value — acts as a *system parameter*: its value is left unspecified, but it must satisfy the refinement predicate. For such a constant, Kind 2 performs a **realizability check**, verifying that the refinement type is realizable, that is, that at least one value satisfies the predicate, so that the parameter can be instantiated:

```
const p: Pos;   -- realizable: some integer is positive
```

By contrast, a free constant whose refinement type is empty is unrealizable:

```
const q: subtype { x: int | x > 0 and x < 0 };   -- unrealizable
```

As with the other realizability checks described above, free-constant realizability is checked with `kind2 <filename> --enable CONTRACTCK`, and a specific constant can be selected using `--lus_main_const <const_name>`.

9.7 Structured types

Refinement types can be arbitrarily nested within structured types (e.g., tuple component types, array and set element types, and map key and value types). For example, consider node N below.

```
type Nat = subtype { x: int | x >= 0 };
const N: Nat;
```

```
node N () returns (my_tuple: [Nat, int]; my_set: set<Nat>;
                   my_array: Nat^N; my_map: map<Nat, Nat>)
...
```

Due to the refinement types, node N carries the following proof obligations: `my_tuple[0] >= 0` (for the tuple's first component type), `forall (e: int) e in my_set => e >= 0` (for the set's element type), `forall (i: int) 0 <= i and i < N => my_array[i] >= 0` (for the array's element type), `forall (k: int) k in my_map => k >= 0` (for the map's key type), and `forall (k: int) k in my_map => my_map[k] >= 0` (for the map's value type). If one has refinement types in node inputs, node locals, or global constants, assumptions or proof obligations (depending on the case) are generated analogously.

10 Enumeration types

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

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.

10.1 Selecting on an enumerated value

A `when` expression evaluates only its selected branch, so a stream can be defined by cases on the value of an enumerated stream, as a Lustre V6 `merge` on an enumerated clock does:

```
o = when c = A then x else w + 1;
```

With more values, a cond block gives one branch per value:

```
cond
  | c = A: o = x;
  | c = B: o = w + 1;
  otherwise: o = x + w;
end
```

A temporal operator or a node call in a branch only advances at the steps at which the branch is selected (see Restart for the correspondence with the clock operators of Lustre).

11 History Types

In order to improve the expressivity of Kind 2's specification language, the tool provides a built-in type constructor that allows users to refer to an unbounded number of previous values of a stream. Specifically, the unary type constructor `history(x)`, that takes a stream `x` of arbitrary type `T` as its single argument, represents the set of all streams `z` of values of type `T` such that at any time `t >= 0`, there exists a `k` in the interval `[0, t]` such that `z(t) = x(k)`.

For instance, given a node with an input stream `x` and an output stream `y`, both with the same type, the property the current value of stream y equals the current value or a previous value of a stream x plus one can't be expressed in Lustre. However, using the type constructor `history`, one can easily express the property as `exists (z: history(x)) y=z+1`.

Notice that `history(x)` denotes a refinement type, suggesting its applicability wherever a type is expected in the model. However, at present, the implementation restricts the use of the type constructor `history` to the type of a quantified variable. We plan to lift this restriction in future versions of Kind 2.

12 Abstract Types

Kind 2 supports Lustre's **abstract types**, which are user-declared types without definitions. Abstract types are declared with the syntax `type <name>`. Below is a simple Lustre file that declares an identity node that takes an input of (abstract) type `T` and returns an output of type `T` equal to the input.

```
type T;
function id_T (x: T) returns (y: T);
let
    y = x;
tel
```

12.1 Domain and quantification

Because an abstract type has no definition, Kind 2 treats it as an *uninterpreted* domain: the only operations available on its values are equality `=` and disequality `<>`. Quantifiers may range over an abstract type, so both `forall (x: T) ...` and `exists (x: T) ...` are allowed. For example:

```
type T;
node N() returns ()
let
  check forall (t: T) t = t;
tel
```

Kind 2 does not assume that the domain of an abstract type is infinite; it may contain any number of values, whether finite or infinite. As a consequence, an assumption that constrains an abstract type to finitely many values is now consistent. For instance, the assumption below restricts `T` to a single value:

```
type T;
function id_T (x: T) returns (y: T);
con
    assume forall (v1: T) (forall (v2: T) v1 = v2);
noc
let
```

```
    y = x;
    check "P1" exists (v: T) y = v;
    check "P2" exists (v1: T) exists (v2: T) v1 <> v2;
tel
```

Under this assumption `P1` holds, because `y` is itself a value of `T`, while `P2`, which asserts that `T` contains two distinct values, is falsified.

This behavior changed after Kind 2 v2.3.0. In v2.3.0 and earlier, abstract types were assumed to have infinite domains, and quantifying over a variable of an abstract type was rejected during type checking. Consequently, an assumption constraining an abstract type to a finite domain (such as the one above) was inconsistent.

13 Polymorphic User Types

Kind 2 supports **polymorphic user types**, which are user-defined types that contain type parameters. An example is a polymorphic user-defined `Pair` type, declared as `type Pair<T; U> = [T, U];`.

A polymorphic user-defined type `T` is instantiated with `T<...>` syntax (analogous to polymorphic nodes and node calls) as in the following examples.

```
type Pair<T; U> = [T, U];

node SwapIntBool(x: Pair<int; bool>) returns (y: Pair <bool; int>)
let
    y = '(x[1], x[0]);
tel

node SwapGeneric<T; U>(x: Pair<T; U>) returns (y: Pair <U; T>)
let
    y = '(x[1], x[0]);
tel
```

In other words, `Pair` (or any other user-defined polymorphic type) can be viewed as as a **type constructor** which takes types as inputs and returns a type.

14 Tuples

Tuples are constructed with the syntax `'(x1, ..., xn)` and destructed with the syntax `t[idx]`, where `idx` is some concrete natural number that is in range (with `0`-based indexing).

```
type my_tuple = [int, bool, real];
node n (x : my_tuple) returns (y : my_tuple)
let
  y = '(0, false, x[2]);
tel
```

This syntax changed after Kind 2 v2.3.0. In v2.3.0 and earlier, tuples were constructed with curly braces, `{x1, ..., xn}`, and a component was accessed with the `.%` operator, as in `t.%idx` (with `0`-based indexing); nested accesses were chained, as in `t.%1.%0`. For example, the definition `y = '(0, false, x[2]);` above would previously have been written `y = {0, false, x.%2};`.

The change was motivated by the introduction of set data types: the `{...}` notation is now used for set constructors.

14.1 Element update

A new tuple that is identical to an existing one except at selected positions can be constructed with the *element update* syntax `t[idx := v]`. It denotes a copy of the tuple `t` in which the component at position `idx` is replaced by `v`; the original tuple `t` is not modified. As with destruction, `idx` must be a concrete natural number that is in range (with `0`-based indexing).

```
type MyPair = [int, bool];
node n (p1 : MyPair) returns (p2 : MyPair)
let
  p2 = p1[1 := true];
tel
```

Here `p2` is equal to `p1` except that its second component (index `1`) is `true`. Several components can be updated at once by separating the individual updates with semicolons, as in `p1[0 := 0; 1 := false]`.

15 Sets

Set types have the syntax `set<T>`, where `T` is any type that does not contain sets, maps, or arrays. For example `set<int>` denotes (streams of) sets of integers, and `set<[bool, int]>` denotes (streams of) sets of Boolean and integer pairs.

Set literals can be constructed with curly braces with comma-separated elements, e.g. `{ 1 }`, `{ 1, 2, 3 }`, and `{ x, 4 }` (assuming `x` has type `int`). Type annotations are *required* for empty sets (e.g. `{}@<int>`).

The built-in set operators are **set union** (denoted by `+`), **set intersection** (denoted by `*`), and **set difference** (denoted by `-`), and **set membership** (denoted by `in`). All are infix and take the expected semantics; see below for an example.

Sets also support **structural equality** (denoted by `=`) and **structural disequality** (denoted by `<>`). Two sets are structurally equal when they contain exactly the same elements, regardless of how they were constructed (e.g., `{ 1, 2 } = { 2, 1, 1 }` is valid).

Two sets of the same type can also be compared with the **subset** (denoted by `<=`), **proper subset** (denoted by `<`), **superset** (denoted by `>=`), and **proper superset** (denoted by `>`) operators. `s1 <= s2` holds when every element of `s1` is also an element of `s2`, and `s1 < s2` holds when, in addition, the two sets are not equal (e.g., `{ 1, 2 } < { 1, 2, 3 }` and `{ 1, 2 } <= { 2, 1, 1 }` are valid, but `{ 1, 2 } < { 2, 1, 1 }` is not). `s1 >= s2` and `s1 > s2` are `s2 <= s1` and `s2 < s1`, respectively.

```
node N (s1, s2: set<int>) returns (out: set<int>)
let
  out = s1 * { 1, 2, 3 } + {}@<int>;

  check forall (i: int) not (i in s1 + s2) = (not (i in s1) and not (i in s2));
  check forall (i: int) (i in s1 - s2) = (i in s1 and not (i in s2));
  check forall (i: int) i in out => (i = 1 or i = 2 or i = 3);

  check out <= { 1, 2, 3 };
  check s1 * s2 <= s1 and s1 <= s1 + s2;
```

```
  check {}@<int> <= s1;
tel
```

16 Maps

Map types have the syntax `map<K, V>` (or `map<K; V>`), where `V` is any type, and `K` is any type that does not contain sets, maps, or arrays. For example `map<int, int>` denotes (streams of) maps of integers to integers, and `map<[bool, int], real>` denotes (streams of) maps of Boolean and integer pairs to reals.

Map literals can be built with the constructor `map[k1 := v1; ...; kn := v2]` which creates a map with keys `k1` through `kn`, each mapping to its corresponding value. For example, `map[0 := 0; 1 := 1]`, `map[{ '(false, 0) } := 0.0]`, and `map[x := y]` are all valid map literals. Type annotations are *required* for empty maps (e.g. `map[]@<int, int>`).

The built-in map operators are **map insertion/update** (denoted by `m[k1 := v1; ...; kn := vn]` for map expression `m`), **map index access** (denoted by `m[k]`), **map subtraction** (denoted by `m - s` for map expression `m` and set expression `s` whose elements have the same type as the keys of `m`, which removes from `m` all keys contained in `s`), and **map (key) membership** (denoted by `in`). Indexing a map with `m[k]` returns the associated value for `k` in map `m`. If `m` does not have a binding for `k`, then the output of the operation is unconstrained. However, the operation is still *functional* in the sense that indexing a map will always yield the same value for a fixed key and timestep (i.e., for all maps `m` and keys `k` of the proper type, `m[k] = m[k]` is valid, even if key `k` is not in the map `m`). Otherwise, the operators all take the expected semantics.

Maps also support **structural equality** (denoted by `=`) and **structural disequality** (denoted by `<>`). Two maps are structurally equal when they bind exactly the same keys to the same values, regardless of how they were constructed (e.g., `map[0 := 0; 1 := 1] = map[1 := 1; 0 := 0]` is valid).

See below for an example.

```
node N (inp: map<int, int>) returns (out, out2, out3: map<int, int>)
let
  out = inp[0 := 0; 1 := 1; 2 := 2];
  out2 = map[0 := 0; 1 := 1];
  out3 = out - { 0, 1 };
```

```
  check 0 in out;
  check out[0] = 0;
  check forall (i: int) not (i in { 0, 1, 2 }) => out[i] = inp[i];
  check forall (i: int) not (i = 0 or i = 1) => not (i in out2);
  check forall (i: int) (i in out3) = (i in out and not (i in { 0, 1 }));
tel
```

17 Subrange types

Subrange types are types of the form `subrange [LB, UB] of int` denoting user-specified integer ranges, where `LB` and `UB` are the lower and upper bound (respectively) of the range (inclusive). In the simplest case, both bounds are concrete integer literals; for example, `subrange [0, 10] of int` denotes the type of streams of integers in the range `0` through `10`, inclusive. Bounds may also be symbolic constant expressions (see Symbolic bounds below).

Subranges may be unbounded on one side, denoted by using a `*` in lieu of a concrete integer for either the upper or lower bound (but not both). For example, `subrange [0, *] of int` denotes the type of streams of nonnegative integers, while `subrange [*, -1] of int` denotes the type of streams of negative integers.

Subrange types can be viewed as particular instances of refinement types: a subrange type on an input variable or free constant can be viewed as an assumption, while a subrange type on an output variable, local variable, or defined constant can be viewed as a proof obligation. For example, consider the following Kind 2 input.

```
type Pos = subrange [1, *] of int;
type Neg = subrange [*, -1] of int;
const C: Pos;

node N(arr: int^C; x: Pos) returns (out: Neg)
let
  out = x - arr[0];
tel
```

Above, `C` and `x` are assumed to be positive integers. However, Kind 2 will generate a proof obligation that `out` is negative, which fails.

17.1 Symbolic bounds

The bounds `LB` and `UB` need not be concrete integer literals: each may also be a *symbolic* constant expression of type `int`, that is, an expression built from integer constants. For example, given a constant `N`, both `subrange [0, N] of int` and

`subrange [1, N-1] of int` are valid subrange types. This is convenient, for instance, to describe the valid indices of an array whose length is a symbolic constant.

```
const N: int;
type Index = subrange [0, N-1] of int;

node M(a: int^N; i: Index) returns (out: int)
let
  out = a[i];
tel
```

When both bounds are concrete integers, Kind 2 checks at compile time that `LB` does not exceed `UB`, rejecting the type otherwise. When at least one bound is symbolic, this check cannot be carried out statically. Instead, following the refinement-type view described above, the range constraint becomes an *assumption* when the symbolic subrange types an input or free constant, and a *proof obligation* when it types an output, local variable, or defined constant.

18 Records

A record type groups a fixed set of named *fields*, each with its own type. Record types are introduced as type aliases, with the `struct` keyword followed by the fields between curly braces, separated by semicolons:

```
type rat = struct { n: real; d: real };
```

A record value is constructed by giving a value to each field, using the type name followed by the field assignments (note that fields are assigned with `=`):

```
r = rat { n = 1.0; d = 2.0 };
```

The field `f` of a record `r` is accessed with the dot syntax `r.f`. For instance, `r.n` evaluates to `1.0` for the record `r` above.

18.1 Element update

A new record that is identical to an existing one except at selected fields can be constructed with the *element update* syntax `r[f := v]`. It denotes a copy of the record `r` in which field `f` is replaced by `v`; the original record `r` is not modified. For example,

```
type rat = struct { n: real; d: real };
node x (r: rat) returns (y: rat)
let
  y = r[n := r.n * 2.0];
tel
```

defines `y` to be equal to `r` except that its `n` field is doubled. Several fields can be updated at once by separating the individual updates with semicolons, as in `r[n := 1.0; d := 2.0]`.

19 Algebraic Datatypes

Kind 2 supports **algebraic datatypes** (ADTs). An ADT packages together a fixed set of *constructors*, each of which may carry zero or more named *fields*.

19.1 Declarations

An ADT is introduced with the `datatype` keyword:

```
datatype Shape =
  | Circle (radius: real)
  | Rectangle (width: real, height: real)
  | Point;
```

Each constructor is separated by `|`. A constructor with no fields (e.g. `Point`) is called *nullary*. Fields are listed inside parentheses and separated by commas.

Multiple ADTs can be declared in a single file, and they may be **recursive**:

```
datatype List = Cons (head: int, tail: List) | Nil;

datatype Nat = Succ (pred: Nat) | Zero;
```

19.2 ADT Terms

A value of an ADT is built by applying a constructor to its field values:

```
datatype Shape = Circle (radius: real) | Rectangle (width: real, height: real);

node main(r: real) returns (s: Shape);
let
  s = Circle(r);
tel
```

A nullary constructor is written without parentheses:

```
datatype Color = Red | Green | Blue;
const c: Color = Red;
```

ADT equality (`=` and `<>`) is structurally defined: two values are equal if and only if they were built with the same constructor and all their fields are pairwise equal (within this given constructor).

19.3 Pattern Matching

The primary way to inspect an ADT value is with match expressions of the form `match ... with ... end`:

```
datatype Color = Red | Green | Blue;

node main(c: Color) returns (ok: bool);
let
  ok = match c with
    | Red   : true
    | Green : false
    | Blue  : true
  end;
tel
```

Each arm `| C (x1, ..., xn) : e` introduces fresh pattern variables `x1, ..., xn` that are bound to the fields of constructor `C` and are in scope only within `e`. Nullary constructors are matched without parentheses. Kind 2 checks pattern matching statically: it rejects a `match` that leaves some value of the scrutinee's type unmatched, and one with an arm that no value can reach because the preceding arms already cover it.

Nested patterns are supported—a field position can itself be a constructor pattern:

```
datatype List = Cons (head: int, tail: List) | Nil;

-- Match the first two elements
out = match inp with
  | Cons (i, Cons (j, _)) : i + j
  | Cons (i, Nil)         : i
```

```
  | Nil                   : 0
end;
```

A wildcard `_` matches any value and binds no name.

19.4 Testers

A **tester** `C?(e)` is a Boolean expression that is `true` if and only if `e` was built with constructor `C`:

```
datatype Shape = Circle (radius: real) | Rectangle (width: real, height: real);

node main(s: Shape) returns (is_circle: bool);
let
  is_circle = Circle?(s);
tel
```

Testers are convenient in conditions where matching on field values is not required:

```
datatype Option<T> = None | Some (val: T);

node main(x: Option<int>) returns (ok: bool);
let
  ok = Some?(x) and then x.val > 0;
tel
```

19.5 Selectors

A **selector** `e.f` extracts field `f` from an ADT value `e`. The field `f` must belong to exactly one constructor of `e`'s type. The selector is *partial*: its result is well-defined only when `e` was built with the constructor that has field `f`. Kind 2 generates a **proof obligation** for each selector use, requiring that the correct constructor is active at the point of use. For example, `s.radius` generates an obligation that `Circle?(s)` holds. The obligation is reported as a property named `Selector[L<line>C<column>]`, positioned at the `.` of the selector. When the obligation does not hold, the value of the selector is arbitrary but fixed for a given ADT value: two reads of the same value agree, and equal values give equal results.

19.5.1 Discharging the obligation

To discharge the proof obligations, one must only use selectors in contexts where the constructor of the term is known. For example, the usage of a tester `C?(t)` in an assumption, the antecedent of a lazy implication `==>`, or the guard of a `when` block is sufficient to prove the well-foundedness of the corresponding selector in the node/function body, lazy implication antecedent, or `then` branch expression, respectively. Only the *lazy* forms guard a selector. The branches of `if ... then ... else ...`, the operands of `and`, `or` and `=>`, and the operands of an `arrow` are all evaluated unconditionally, so a tester there does not discharge an obligation.

```
datatype Shape = Circle (radius: real) | Rectangle (width: real, height: real);

node is_large_circle(s: Shape) returns (ok: bool);
let
  ok = Circle?(s) ==> s.radius > 10.0;
tel
```

19.6 Polymorphic ADTs

ADT declarations may have **type parameters**, making them polymorphic:

```
datatype Option<T> = None | Some (val: T);
datatype Either<A; B> = Left (left: A) | Right (right: B);
```

A polymorphic ADT is instantiated by supplying type arguments:

```
datatype Option<T> = None | Some (val: T);

node main(i: int; r: real) returns (ok: bool);
var xi: Option<int>;
    xr: Option<real>;
let
  xi = Some(i);
  xr = Some(r);
  ok = Some?(xi) and Some?(xr);
tel
```

20 JSON / XML Output

Kind 2 can output its results in two structured formats: JSON and XML. They facilite the processing of Kind 2's results by external tools. The next sections describe each of these output formats in detail.

20.1 JSON format

The JSON output is activated by running Kind 2 with the `-json` option. Its syntax is fully specified by the JSON schema available in the schemas/kind2-output.json file.

The root element of a JSON output document is either a Log Object if Kind 2 terminates early with an error, or an array of Results Objects if Kind 2 succeeds generating some result. Every Results Object (including Log Object) is identified and distinguished from other Results objects by a property of type string called `objectType`.

In a successful execution, a Kind2 Options Object specifies the options used by the tool, and any Log message is added to the array as it is written. When Kind 2 is run as an interpreter, the array includes one Execution Object that contains a description of the computed values for the output and state variables. Otherwise, Kind 2 works as a model checker and performs a series of analyses. The beginning of a main analysis is indicated by an AnalysisStart Object, and its end by an AnalysisStop Object. Within these delimiters, a Property Object describes the result for a particular property of the input model under the parameters of the analysis. When the verbose mode is enabled, statistics and progress info of the analysis is also recorded along through Stat and Progress objects.

Similarly to main analyses, when a post-analysis is enabled, the beginning of the post-analysis is indicated by an PostAnalysisStart Object, and its end by an PostAnalysisEnd Object.

20.1.1 Incremental JSON format

The incremental JSON output is activated by running Kind 2 with the `-ijson` option. It contains the same Results Objects as the regular JSON output, but prints them as a

sequence of independent JSON objects rather than as elements of a single enclosing array.

With the `-json` option, the output is not a complete, parsable JSON document until Kind 2 terminates and closes the root array. The `-ijson` option emits each object as a complete JSON value as soon as it becomes available. This allows external tools to process Kind 2's output incrementally without waiting for the entire analysis to finish.

20.1.2 Log Object

A `Log` object records an informative message from the tool. The value of its `objectType` property is `log`. The list of properties of a `Log` object are:

Key	Type	Description
`level`	`string`	A level that gives a rough guide of the importance of the message. Can be `fatal`, `error`, `warn`, `note`, `info`, `debug`, or `trace`.
`source`	`string`	The name of the Kind 2 module which wrote the log.
`file`	`string`	Associated input file, if any.
`line`	`integer`	Associated line in the input file, if any.
`column`	`integer`	Associated column in the input file, if any.
`value`	`string`	The log message.

20.1.3 Results Objects

A `Result object` can be one of the following objects: a Log Object, a Kind2 Options Object, an AnalysisStart Object, an AnalysisStop Object, a Property Object, a Stat Object, a Progress Object, a PostAnalysisStart Object, or a PostAnalysisEnd Object.

20.1.4 Kind2 Options Object

A `Kind2 options` object describes the options used by the tool in the current execution. The value of its `objectType` property is `kind2Options`. The list of properties of a `Kind2 options` object are:

Key	Type	Description
`enabled`	`array`	List of Kind 2 module names that are enabled.
`timeout`	`number`	The wallclock timeout used for all the analyses.
`bmcMax`	`integer`	Maximal number of iterations for BMC and K-induction.
`compositional`	`boolean`	Whether compositional analysis is enabled or not.
`modular`	`boolean`	Whether modular analysis is enabled or not.

20.1.5 AnalysisStart Object

An `AnalysisStart` object indicates the beginning of a main analysis. The value of its `objectType` property is `analysisStart`. The list of properties of an `AnalysisStart` object are:

Key	Type	Description
`top`	`string`	Name of the current top-level component.
`concrete`	`array`	Names of the subcomponents whose implementation is used in the analysis.
`abstract`	`array`	Names of the subcomponents whose contract is used in the analysis.
`assumptions`	`array`	Array of pairs (name of subcomponent, number of considered invariants).

An `AnalysisStart` object that begins a check of the contract checker (`--enable CONTRACTCK`) has `top` and the following property instead of `concrete`, `abstract` and `assumptions`:

Key	Type	Description
`context`	`string`	What is checked for `top`: `type`, `contract`, or `environment`.

20.1.6 AnalysisStop Object

An `AnalysisStop` object indicates the end of a main analysis. The value of its `objectType` property is `analysisStop`. No properties are associated.

20.1.7 Property Object

A `Property` object describes the result for a particular property of the input model. The result should be considered in the context of the analysis in which the property object is contained. The value of its `objectType` property is `property`. The list of properties of an `AnalysisStart` object are:

Key	Type	Description
`name`	`string`	Property identifier or description.
`scope`	`string`	Name of the component where the property was analyzed.
`line`	`integer`	Associated line in the input file, if any.
`column`	`integer`	Associated column in the input file, if any.
`source`	`string`	Origin of the property. Can be `Assumption` if it comes from an assumption check, `Guarantee` if it comes from the check of a guarantee, `Ensure` if it comes from a check of a require-ensure clause in a contract mode, `OneModeActive` if it comes from an exhaustiveness check of the state space covered by the modes of a contract, and `PropAnnot` if it comes from the check of a property annotation.
`runtime`	`object`	The runtime of the analysis (in seconds), and whether the timeout expired
`k`	`integer`	The value of `k` in a k-inductive proof, if any.
`trueFor`	`integer`	The largest value of `k` for which the property was proved to be true, if any.
`answer`	`object`	

Key	Type	Description
		The `source` of the answer, and the result `value` of the check. The result can be `valid`, `falsifiable`, or `unknown`.
`counterExample`	`object`	Counterexample to the property satisfaction (only available when `answer` is `falsifiable`). It describes a sequence of values for each stream that leads the system to the violation of the property. It also gives the list of contract modes that are active at each step, if any.

20.1.8 Stat Object

An `Stat` object provides statistics info about the current analysis. The value of its `objectType` property is `stat`. The list of properties of a `Stat` object are:

Key	Type	Description
`source`	`string`	Name of the Kind 2 module which reported the info.
`sections`	`array`	List of `statSection` objects, each of them with a section `name` and a list of `statItem` objects. Each `statItem` has a `name`, a `type`, and a `value`. See schemas/kind2-output.json for further details.

20.1.9 Progress Object

An `Progress` object reports the current value of `k` for k-inductive-based analyses. The value of its `objectType` property is `progress`. The list of properties of a `Progress` object are:

Key	Type	Description
`source`	`string`	Name of the k-inductive-based analysis.
`k`	`integer`	Value for `k`.

20.1.10 PostAnalysisStart Object

An `PostAnalysisStart` object indicates the beginning of a post-analysis. The value of its `objectType` property is `postAnalysisStart`. The list of properties of an `PostAnalysisStart` object are:

Key	Type	Description
`name`	`string`	Name of the post-analysis

The post-analyses currently available are Test Generation (`testgen`), Proof Certificates (`certification`), Contract Generation (`contractgen`), Invariant Printing (`invprint`), and Inductive Validity Core (`ivc`).

20.1.11 PostAnalysisEnd Object

An `PostAnalysisEnd` object indicates the end of a post-analysis. The value of its `objectType` property is `postAnalysisEnd`. No properties are associated.

20.1.12 Execution Object

An `Execution` object describes the sequences of values for the output and state variables of an input model computed from its simulation (see the interpreter mode). The value of its `objectType` property is `execution`. It only has one object property called `trace` which follows the same format than property `counterExample` in Property Object.

20.1.13 ModelElementSet Object

A `ModelElementSet` object describes a set of model elements (a model element can be an equation, a node call, an assumption, a guarantee, etc). It is used to describe a core that we can get from an Inductive Validity Core (`ivc`) or Minimal Cut Set (`mcs`) analysis. The result should be considered in the context of the analysis or post-analysis in which the ModelElementSet object is contained. The value of its `objectType` property is `modelElementSet`.

Key	Type	Description
`class`	`string`	

Key	Type	Description
		Class of the core. Can be `must`, `must complement`, `ivc`, `ivc complement`, `mcs` or `mcs complement`.
`size`	`integer`	Number of model elements in the core.
`property`	`string`	The property associated with the core. If all properties are considered, this field is missing.
`runtime`	`object`	The runtime for computing the core (in seconds).
`nodes`	`array`	For each node, contains an object that enumerates the model elements of the node that are part of the core.
`counterExample`	`object`	Counterexample to the property satisfaction (only when relevant, that is, when class is `mcs` or `mcs complement`). See the `property` object for more info.

20.2 XML format

The XML output is activated by running Kind 2 with the `-xml` option. Its syntax is fully specified by the XML schema available in the schemas/kind2-output.xsd file.

The root element of a XML output document is either a Log Element if Kind 2 terminates early with an error, or a Results Element if Kind 2 succeeds generating some result.

20.2.1 Log Element

A `Log` element is a simple element that records an informative message from the tool. The list of attributes of a `Log` element are:

Attribute	Base Type	Description
`class`	`xs:string`	A level that gives a rough guide of the importance of the message. Can be `fatal`, `error`, `warn`, `note`, `info`, `debug`, or `trace`.

Attribute	Base Type	Description
`source`	`xs:string`	The name of the Kind 2 module which wrote the log.
`line`	`xs:integer`	Associated line in the input file, if any.
`column`	`xs:integer`	Associated column in the input file, if any.

20.2.2 Results Element

A `Results` element is a sequence of zero or more of the following elements: a Log Element, an AnalysisStart Element, an AnalysisStop Element, a Property Element, a Stat Element, a Progress Element, a PostAnalysisStart Element, a PostAnalysisEnd Element, or an Execution Element.

The list of attributes of a `Results` element are:

Attribute	Base Type	Description
`enabled`	`xs:string`	List of comma-separated Kind 2 enabled module names.
`timeout`	`xs:decimal`	The wallclock timeout used for all the analyses.
`bmc_max`	`xs:integer`	Maximal number of iterations for BMC and K-induction.
`compositional`	`xs:boolean`	Whether compositional analysis is enabled or not.
`modular`	`xs:boolean`	Whether modular analysis is enabled or not.

20.2.3 AnalysisStart Element

An `AnalysisStart` element is an empty element that indicates the beginning of a main analysis. The list of attributes of an `AnalysisStart` element are:

Attribute	Base Type	Description
`top`	`xs:string`	Name of the current top-level component.
`concrete`	`xs:string`	

Attribute	Base Type	Description
		Names of the subcomponents whose implementation is used in the analysis (comma-separated list).
`abstract`	`xs:string`	Names of the subcomponents whose contract is used in the analysis (comma-separated list).
`assumptions`	`xs:string`	Comma-separated list of pairs (subcomponent name, number of considered invariants).

20.2.4 AnalysisStop Element

An `AnalysisStop` element is an empty element that indicates the end of a main analysis. No attributes.

20.2.5 Property Element

A `Property` element describes the result for a particular property of the input model. The result should be considered in the context of the analysis in which the property element is contained. The list of attributes of a `Property` element are:

Attribute	Base Type	Description
`name`	`xs:string`	Property identifier or description.
`scope`	`xs:string`	Name of the component where the property was analyzed.
`file`	`xs:string`	Associated input file, if any.
`line`	`xs:integer`	Associated line in the input file, if any.
`column`	`xs:integer`	Associated column in the input file, if any.
`source`	`xs:string`	Origin of the property. Can be `Assumption` if it comes from an assumption check, `Guarantee` if it comes from the check of a guarantee, `Ensure` if it comes from a check of an ensure clause in a contract mode, `OneModeActive` if it comes from an exhaustiveness check of the state space covered by the modes of a

Attribute	Base Type	Description
		contract, and `PropAnnot` if it comes from the check of a property annotation.

A `Property` element contains one `Answer` element, which includes the result for the property check (`valid`, `falsifiable`, or `unknown`), and identifies the Kind 2 module responsible for the answer. If the result is `valid`, or `falsifiable`, it also contains a `Runtime` element, which reports the runtime of the analysis (in seconds), and whether the timeout expired or not. If the result is `valid`, a `K` element gives the value of `k` for which the property was proved valid. If the result is `falsifiable`, a `Counterexample` element describes a sequence of values for each stream that leads the system to the violation of the property. It also gives the list of contract modes that are active at each step, if any. If the result is `unknown`, the `Property` element may contain a `TrueFor` element specifying the largest value of `k` for which the property was proved to be true.

20.2.6 Stat Element

An `Stat` element provides statistics info about the current analysis. It has only one attribute of type `xs:string`, `source`, which is the name of the Kind 2 module which reported the piece of information. Its content consists in one or more `Section` elements. Each section has one `name` element, and one or more `item` elements. Each `item` element has one `name` element, and either a `value` element or a `valuelist` element. A `valuelist` has one or more `value` elements, and each `value` element has a `type` attribute (currently `int` or `float`), and its content is the actual value.

20.2.7 Progress Element

A `Progress` element is a simple element that reports the current value of `k` for a k-inductive-based analysis. It has only one attribute of type `xs:string`, `source`, which is the name of the k-inductive-based analysis.

20.2.8 PostAnalysisStart Element

An `PostAnalysisStart` element is an empty element that indicates the beginning of a post-analysis. It has only one attribute of type `xs:string`, the `name` of the post-analysis. The post-analyses currently available are Test Generation (`testgen`), Proof

Certificates (`certification`), Contract Generation (`contractgen`), Invariant Printing (`invprint`), and Inductive Validity Core (`ivc`).

20.2.9 PostAnalysisEnd Element

An `PostAnalysisEnd` element is an empty element that indicates the end of a post-analysis. No attributes.

20.2.10 Execution Element

An `Execution` element describes the sequences of values for the output and state variables of an input model computed from the simulation of its execustion (see the interpreter mode).

20.2.11 ModelElementSet Element

A `ModelElementSet` element describes a set of model elements (a model element can be an equation, a node call, an assumption, a guarantee, etc). It is used to describe a core that we can get from an Inductive Validity Core (`ivc`) or Minimal Cut Set (`mcs`) analysis. The result should be considered in the context of the analysis or post-analysis in which the ModelElementSet element is contained. The list of attributes of a `ModelElementSet` element are:

Attribute	Base Type	Description
`class`	`string`	Class of the core. Can be `must`, `must complement`, `ivc`, `ivc complement`, `mcs` or `mcs complement`.
`size`	`integer`	Number of model elements in the core.
`property`	`string`	The property associated with the core. If all properties are considered, this attribute is missing.

A `ModelElementSet` element contains one `Runtime` element, which indicates the runtime for computing the core. It also contains a sequence of `Node` elements, each one enumerating the model elements in that node that are part of the core. When relevant, it can also contain a `Counterexample` element (see the `Property` element for more info).

21 Exit codes

Since version 1.9.0, Kind 2 returns the standard exit code `0` for success, and a non-zero exit code to indicate an error, or an unsuccessful analysis result. To force Kind 2 to use only non-zero exit codes for errors, pass the option `--exit_code_mode only_errors`. The precise meaning of the exit codes are described in section Code Convention. For information on the old convention, see section Former Convention.

21.1 Code Convention

With the default settings, Kind 2 performs a single, monolithic analysis of the main node. If Kind 2 proves all invariant properties valid and all reachability properties reachable, then it returns (exit code) `0`. If no properties are disproven, but some properties could not be proven (e.g. due to a timeout), Kind 2 returns `30`. When Kind 2 disproves one or more properties, it returns `40`.

In modular mode, the properties of all nodes are checked bottom-up. Moreover, when compositional analysis is enabled too, the same node may be analyzed several times with different levels of abstraction (see section Refinement in compositional and modular analyses for details). In this case, Kind 2 returns `40` if one or more properties were disproven in *any* analysis. It returns `30` if no properties were disproven, but some nodes were not analyzed (e.g. due to a timeout) or some properties could not be proven. It returns `0` if all properties were proven for all nodes in every analysis.

When contracts of imported nodes are checked for realizability, Kind 2 also reports an exit status following a similar convention. If all the contracts are proven realizable, it returns `0`. If some contract is proven unrealizable, it returns `40`. When no contract is proven unrealizable, but some contract could not be proven realizable, it returns `30`.

If Kind 2 detects a general error, it returns `1`. When the error is related to an incorrect command-line argument, it returns `2`. If Kind 2 detects a parse error, it returns `3`. If Kind 2 cannot find an SMT solver on the PATH, it returns `4`. When an unknown or unsupported version of an SMT solver is detected, it returns `5`.

21.2 Former Convention

Version 1.8.0 and earlier were not following the POSIX convention of returning `0` for success. When all properties were proven, Kind 2 returned `20`. If some property was disproven, it returned `10`. If no properties were disproven, but some result was unknown, it returned `0`. Moreover, Kind 2 returned `2` for any error.

22 Contract Semantics

22.1 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 Contracts 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 all of its executions satisfy the temporal LTL formula

$$\Box A \Rightarrow \Box G$$

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
```

22.2 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.

22.2.1 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

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

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) ;
con
  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 ;
  ) ;
noc
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.

22.2.2 Defensive checks

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

$$(A, G, \{(R_i, E_i)\})$$

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

$$\mathsf{one_mode_active} = \bigvee_i (\bigwedge_j r_{ij})$$

is an invariant for the system

$$A \wedge G \wedge (\bigwedge r_i \Rightarrow \bigwedge e_i)$$

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 and provide a counterexample.

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.

In addition, Kind 2 checks that all modes are reachable in the system. In other words, Kind 2 also checks that for each mode there exists a reachable state satisfying the conjunction of its requires. This lets you know whether the mode implication is vacuously true or not.

When the node associated to the contract has a body (it is not imported), the check will be performed twice. First, considering only the information of the contract. Then, considering the equations of the body too.

Notice that when running Kind 2 in modular mode, the reachability check is performed locally to a node without taking call contexts into account; only the specified assumptions are considered.

You can disable this check by passing `--check_nonvacuity false` to Kind 2, or by suppressing all reachability checks (`--check_reach false`).

23 Post Analysis 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,
- test generation,
- contract generation,
- assumption generation,
- invariant printing, and
- inductive validity core generation

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.

23.1 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	Conditions	Notes
certification	last analysis proved the system safe	will fail if node is partially defined
test generation	system has a contract with more than one mode	
and the last analysis proved the system safe		
contract generation		experimental
assumption generation	last analysis falsified some property	generates non-temporal constraints
invariant printing		
inductive validity		
core generation	last analysis proved the system safe	

24 Test Generation

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 (see Contract Semantics), and more precisely on the notion on *mode*.

24.1 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:

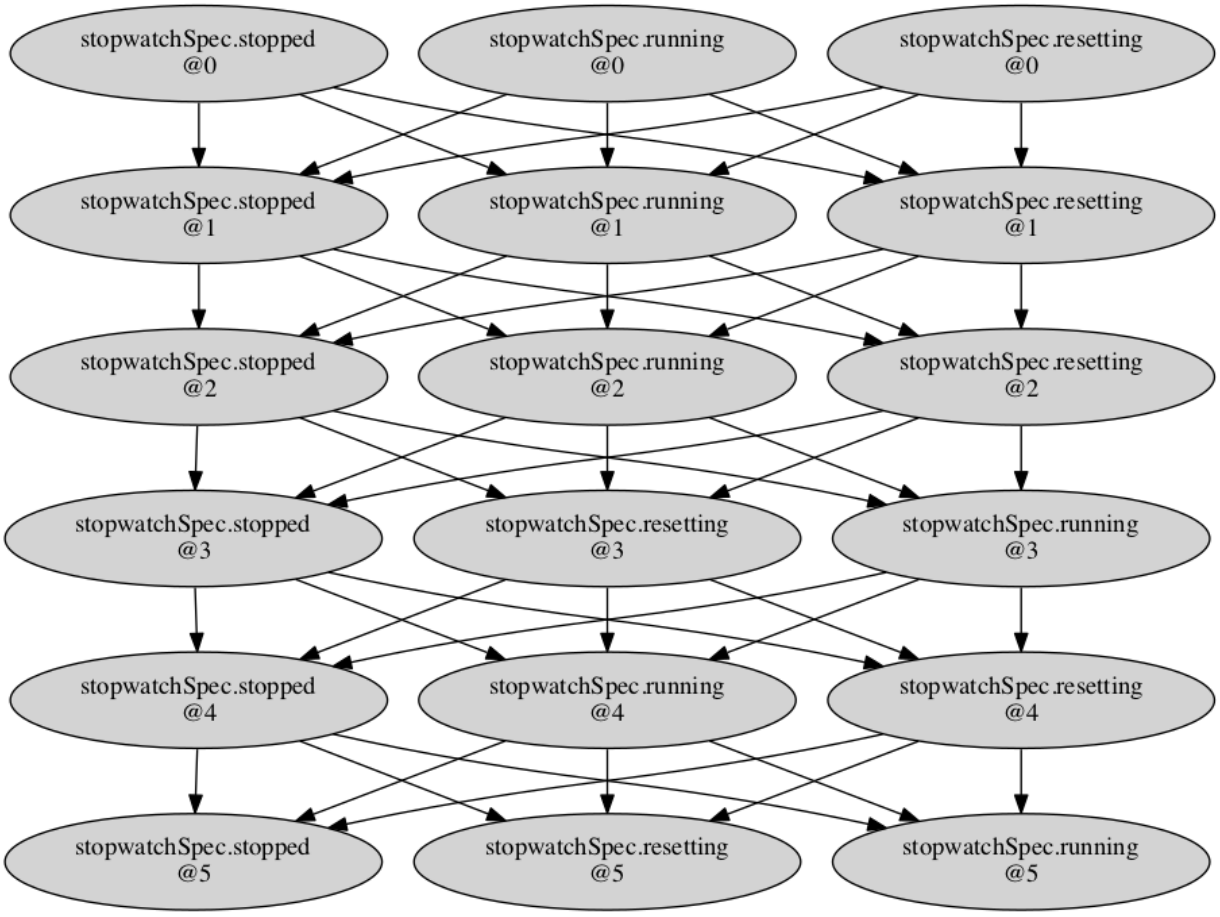
```
contract stopwatchSpec ( tgl, rst : bool ) returns ( c : int ) ;
let
  var on: bool = tgl -> (pre on and not tgl) or (not pre on and tgl) ;
  assume not (rst and tgl) ;
  guarantee c >= 0 ;
```

```
  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 ) ;
con
  import stopwatchSpec ( toggle, reset ) returns ( count ) ;
noc
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 true --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):

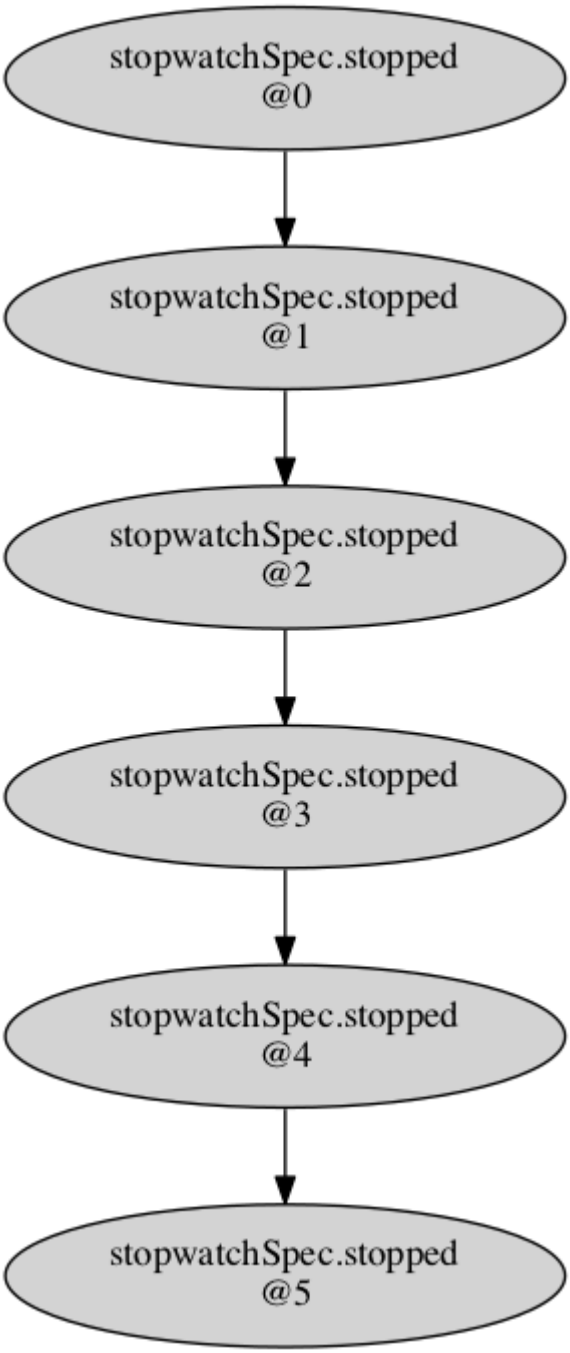


Stopwatch DAG

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:



Stopwatch mistake DAG

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.

24.2 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 JSON file. It is in the same format as the interpreter input format. (See the Interpreter)

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 the B-method for instance.)

24.3 Analyzing the executable

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

For convenience, Kind 2 offers a feature, a contract monitor, which checks whether the output produced by the executable for a given test case respects the contract.

The contract monitor reads the input values of the test case that are fed to the System Under Test (SUT), along with the output values returned by the SUT, and reports the truth values of the guarantees and modes of the original contract. The input format for the contract monitor is the same as the interpreter input format, except that the input includes not only the values of the input variables but also the values of the output variables.

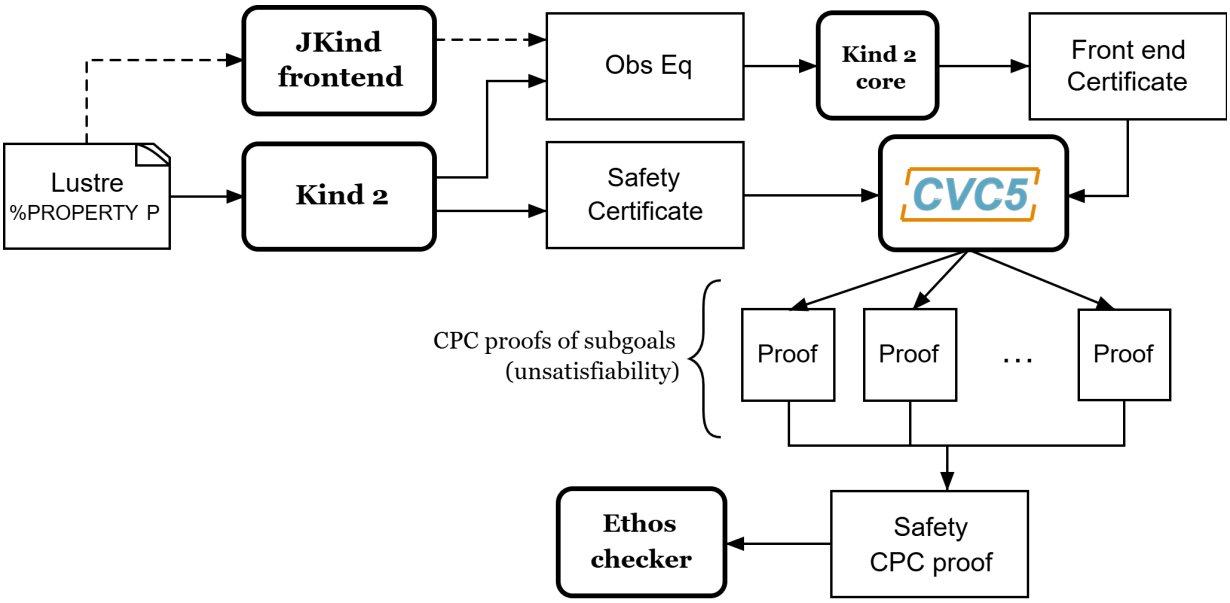
25 Proof Certificates

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 counterexample 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 counterexample 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 Ethos checker.

25.1 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 cvc5, then turned into CPC (Cooperating Proof Calculus) proof objects by collecting cvc5's own proofs and assembling them to form an overall Safety CPC proof that can be efficiently verified by the Ethos proof checker.



Certification process

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.

25.2 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;
      P : bool;
let
  v = a + b ;
  c = 1.0 -> if (pre c) > v then (pre c) else v ;
  P = (a > 0.0 and b > 0.0) => c > 0.0 ;
  --%PROPERTY P;
tel
```

Kind 2 offers the possibility to generate two types of certificates: SMT-LIB 2 certificates and actual proofs in the Safety CPC format, an extension of cvc5 CPC proof format that includes proof rules for invariant and safety properties. It will do so only for systems whose properties (or contracts) are all proven valid.

25.2.1 Requirements

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

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

Safety proof production requires cvc5 (the binary can be specified with `--cvc5_bin`), the Eunoia signatures for CPC proofs, the Eunoia signatures for Safety CPC proofs (included in the Kind 2 distribution), and the Ethos checker for the final proof checking phase.

Ethos checker

A bash script to download and build the Ethos checker is distributed with Kind 2:

```
proofs/get-ethos-checker
```

The script also downloads the cvc5 Eunoia signatures for CPC and generates an easy-to-use bash script (`ethos-check.sh`) to check Safety CPC proofs generated by Kind 2:

```
proofs
|-- get-ethos-checker.sh
|-- bin
    |-- ethos
    |-- ethos-check.sh
|-- signatures
    |-- cpc
      |-- ...
    |-- Safety.eo
```

25.2.2 SMT-LIB 2 certificates

These certificates are always produced but are only used as an intermediate step for CPC 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:

```
Post-analysis: certification

Certificate checker was written in add_two.lus.out/certif/certificate.smt2
Generating frontend eq-observer with jKind ...
Generating frontend certificate
...
Certificate checker was written in add_two.lus.out/certif/FEC.kind2.out/certif/FECC.smt2
```

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

```
add_two.lus.out/certif
|-- certificate_checker
|-- certificate_prelude.smt2
|-- certificate.smt2
|-- FEC.kind2
|-- FEC.kind2.out/certif
    |-- FECC_checker
    |-- FECC_prelude.smt2
    |-- FECC.smt2
    |-- observer_sys.smt2
|-- jkind_sys.smt2
|-- kind2_sys.smt2
|-- observer.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.lus.out/certif/certificate_checker z3
Checking base case
unsat
Checking 1-inductive case
unsat
Checking property subsumption
unsat

> add_two.lus.out/certif/FEC.kind2.out/certif/FECC_checker z3
Checking base case
unsat
Checking 1-inductive case
unsat
Checking property subsumption
unsat
```

25.2.3 Safety CPC proofs

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

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

Successful runs emit outputs that contain lines such as:

```
Post-analysis: certification

Generating proof...

Generating frontend eq-observer with jKind...
...
Generating frontend proof...

Generating safety proof...

Final Safety CPC proof written to add-two.lus.out/certificates.1/safety_proof.cpc
```

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.lus.out/
|-- FEC.kind2
|-- certificates.1
   |-- frontend_base.smt2
   |-- frontend_induction.smt2
   |-- kind2_sys.smt2
   |-- induction.smt2
   |-- frontend_implication.smt2
   |-- obs_phi.smt2
   |-- jkind_sys.smt2
   |-- base.smt2
   |-- implication.smt2
   |-- kind2_phi.smt2
   |-- observer.smt2
|-- proofs.1
   |-- induction.cpc
   |-- base.cpc
   |-- frontend_implication.cpc
   |-- frontend_proof.cpc
   |-- kind_2_proof.cpc
   |-- implication.cpc
   |-- frontend_base.cpc
   |-- frontend_induction.cpc
   |-- safety_proof.cpc
```

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 safety proof is an actual proof, one needs to call the Ethos checker on the generated output, `safety_proof.cpc` , together with the correct signatures:

```
proofs/bin/ethos {--include <cpc signature>}* --include <safety signature> add-two.lus.out/ce
```

or use the convenient bash script generated by `proofs/get-ethos-checker.sh` :

```
proofs/bin/ethos_check.sh add-two.lus.out/certificates.1/safety_proof.cpc
```

The return code for either command execution is `0` when everything was checked correctly.

When the bash script is used and the whole proof is correct, the following line will be displayed:

```
correct
```

When the bash script is used and the proof does not contain a proof of safety, the following line will be displayed

```
; WARNING: Safety proof missing
```

The proof may be correct, but this warning tells you that it does not actually represent a proof of safety.

25.3 Contents of certificates

For a given problem (whose safety property is P), an internal certificate consists in only a pair (k, ϕ) where ϕ 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 ϕ implies P and is *k*-inductive. The Safety CPC proof is a formal proof that P is invariant in the system, using sub-proofs of validity (unsatisfiability) returned by cvc5.

25.4 CPC signature

A proof system is formally defined in Eunoia through *signatures*, which contain a definition of the system's language together with axioms and proof rules. The proof system used by cvc5 is defined over a number of signatures, which are included in its source code distribution. Those relevant to this work include the base signatures `Cpc.eo` and more advanced signatures `CpcExpert.eo`.

cvc5's proof system, CPC, is extended with an additional signature (`Safety.eo`), which defines the Safety CPC proof system. This is 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 Safety CPC encoding will make y and z respectively functions from naturals to reals and integers. So we will use the tuples $(y(0), z(0)), (y(1), z(1)), \ldots$ instead of $(y_0, z_0), (y_1, z_1), \ldots$ where $y_0, y_1, \ldots, z_0, z_1, \ldots$ are (distinct) variables. Correspondingly, our Safety CPC 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 ϕ are parametrized by a single natural variable.

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

$$\begin{aligned} \mathrm{invariant} : \Pi\, \mathrm{I} : \mathbb{N} \to \mathrm{formula}. \\ \Pi\, \mathrm{T} : \mathbb{N} \to \mathbb{N} \to \mathrm{formula}. \\ \Pi\, \mathrm{I} : \mathbb{N} \to \mathrm{formula}.\mathrm{Type} \end{aligned}$$

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, $\mathrm{safe}(I, T, P, I', T', P')$, which can be derived only when $\mathrm{invariant}(I, T, P)$ 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 cvc5.

$$\dfrac{\dfrac{\dfrac{k \in \mathbb{N} \quad \text{SMT}\dfrac{\vdots}{B_k \models \bot} \quad \text{SMT}\dfrac{\vdots}{S_k \models \bot}}{\text{K-IND}\ \mathsf{invariant}(I,T,\phi)} \quad \text{SMT}\dfrac{\vdots}{\phi \models P}}{\text{INVIMPL}\ \mathsf{invariant}(I,T,P)} \quad \dfrac{\dfrac{\text{K-IND}\dfrac{\vdots}{\mathsf{invariant}(I_o,T_o,\phi_o)} \quad \text{SMT}\dfrac{\vdots}{\phi_o \models P_o}}{\text{INVIMPL}\ \mathsf{invariant}(I_o,T_o,P_o)}}{\text{OBSEQ}\ \mathsf{woe}(I,T,P,I',T',P')}}{\text{INV+OBS}\ \mathsf{safe}(I,T,P)}$$

Proof sketch

26 Contract Generation

> **Disclaimer:** 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. Note that the restriction to just input and output variables causes many of the generated invariants to be discarded currently.

27 Invariant Printing

This treatment minimizes the invariants used in the proof of the valid properties, and shows them in the output without logging them on disk.

28 Interpreter

The interpreter is a special mode where Kind 2 reads input values from a file and prints the computed values for the output and local variables of a node at each step. If the Lustre file contains two or more top nodes, a single node must be selected with either the command-line option `--lus_main <node_name>` or a single `--%MAIN` annotation in the Lustre file.

To use the interpreter, run:

```
kind2 --enable interpreter <lustre_file> --interpreter_input_file <input_file>
```

You can specify the number of steps to run with the option `--interpreter_steps <int>`. By default, the number of steps is determined by the input file.

28.1 Structure of the input file

The inputs must be specified in a JSON file.

The overall structure is as follows:

```
[
  {
    "var1": "42",
    "var2": true,
    "var3": "0.5"
  },
  {
    "var1": "24",
    "var2": false,
    "var3": "1.0/2.0"
  }
]
```

The top-level JSON array corresponds to the successive time steps. Each time step is described by a JSON object associating to each input variable its value for this time step.

NOTE: Kind2 also accepts the CSV format for backward compatibility reasons. However, it does not support records, arrays and tuples. Please give your input file the adequate extension (.json or *.csv) in order to indicate to Kind2 which format you are using.*

28.2 Integers and reals

As in the above example, integers and reals should be written as strings in order to avoid a potential loss of precision or an integer overflow while parsing the file. Nevertheless, small integers can be written as native JSON integers without problem.

28.3 Records

Record values can be expressed using a JSON object.

For instance, a variable `c` of type `{ re: real; im: real }` can be assigned as follows:

```
[
  {
    "c": { "re": "-1.0", "im": "0.25" }
  }
]
```

28.4 Arrays

Array values can be expressed using a JSON array.

For instance, a variable `a` of type `bool^3^2` can be assigned as follows:

```
[
  {
    "a": [[true, true, false], [false, true, true]]
  }
]
```

28.5 Tuples

The JSON format does not support tuples by default, so the emelents of the tuple must be named by their index in a JSON object.

For instance, a variable `t` of type `[int, bool, real]` can be assigned as follows:

```
[
  {
    "t": { "0":"36", "1": false, "2":"5.0" }
  }
]
```

Alternatively, a tuple can be expressed using a JSON array.

```
[
  {
    "t": [ "36", false, "5.0" ]
  }
]
```

29 Contract Monitor

The contract monitor is a special mode where Kind 2 reads input and output values from a file and prints a full contract trace of the execution of the top-level node and the subnodes that appear in the contract. The trace includes truth values for assumptions, guarantees, mode requires, and mode ensures.

To use the contract monitor, run:

```
kind2 --enable contract_monitor <lustre_file> --monitor_trace_file <input_file>
```

You can specify the number of steps to run with the option `--monitor_steps <int>`. By default, the number of steps is determined by the input file.

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 ) ;
con
  import stopwatchSpec ( toggle, reset ) returns ( count ) ;
noc
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
```

Using a JSON file contianing a full execution trace:

```
[
  {
  "toggle": true,
  "reset": false,
  "count": 1
  },
  {
    "toggle": true,
    "reset": false,
    "count": 1
  },
  {
    "toggle": false,
    "reset": false,
    "count": 1
  },
  {
    "toggle": false,
    "reset": false,
    "count": 1
  },
  {
    "toggle": true,
    "reset": false,
    "count": 2
  },
  {
    "toggle": false,
    "reset": false,
    "count": 3
  }
]
```

```
kind2 --enable contract_monitor stopwatch.lus
--monitor_trace_file stopwatch_trace.json --monitor_steps 6
```

This produces the following output:

```
Execution:
Node stopwatch ()
  == Assumptions ==
  assume[l4c3]          tt    tt    tt    tt    tt    tt
  == Guarantees ==
  guarantee[l5c3]       tt    tt    tt    tt    tt    tt
  == Modes ==
  stopped.requires      ff    tt    tt    tt    ff    ff
  stopped.ensures       ff    tt    tt    tt    ff    ff
  running.requires      tt    ff    ff    ff    tt    tt
  running.ensures       tt    ff    ff    ff    tt    tt
  resetting.requires    ff    ff    ff    ff    ff    ff
  resetting.ensures     ff    ff    ff    ff    ff    ff
  == Inputs ==
  toggle                tt    tt    ff    ff    tt    ff
  reset                 ff    ff    ff    ff    ff    ff
  == Outputs ==
  count                  1     1     1     1     2     3
  == Ghosts ==
  on                    tt    ff    ff    ff    tt    tt
```

30 Inductive Validity Core

The inductive validity core generation is a post-analysis treatement that computes a minimal subset of the model elements (assumptions, guarantees, stateful equations, or node calls) that are sufficient to prove all valid properties.

To enable inductive validity core generation, run

```
kind2 <lustre_file> --ivc true
```

30.1 Options

- `--ivc_category {node_calls|contracts|equations|assertions|annotations}` (default: all categories) – Minimize only a specific category of elements, repeat option to minimize multiple categories
- `--ivc_only_main_node <bool>` (default `false`) – Only elements of the main node are considered in the computation
- `--ivc_all <bool>` (default `false`) – Compute all the Minimal Inductive Validity Cores
- `--ivc_approximate <bool>` (default `true`) – Compute an approximation (superset) of a MIVC. Ignored if `--ivc_all` or `--ivc_must_set` is `true`
- `--ivc_smallest_first <bool>` (default `false`) – Compute a smallest IVC first. If `--ivc_all` is `false`, the computed IVC will be a smallest one
- `--ivc_must_set <bool>` (default `false`) – Compute the MUST set in addition to the IVCs
- `--print_ivc <bool>` (default `true`) – Print the inductive validity core computed
- `--print_ivc_complement <bool>` (default `false`) – Print the complement of the inductive validity core computed (= the elements that were not necessary to prove the properties)
- `--minimize_program {no|valid_lustre|concise}` (default `no`) – Minimize the source Lustre program according to the inductive validity core(s) computed
- `--ivc_output_dir <string>` (default `<INPUT_FILENAME>`) – Output directory for the minimized programs

- `--ivc_uc_timeout <int>` (default `0`) – Set a timeout for each unsat core check sent to the solver
- `--ivc_precomputed_mcs <int>` (default `0`) – When computing all MIVCs, set a cardinality upper bound for the precomputed MCSs (helps prune space of candidates)

30.2 Example

Let's consider the following Lustre code:

```
contract fSpec(u,v: real) returns(r: real);
let
    guarantee r >= 0.0;
    guarantee true -> r >= pre(r);
    guarantee r >= u;
    guarantee r >= v;
tel;

node f(u, v : real) returns (r : real);
con
    import fSpec(u,v) returns (r) ;
noc
var m1,m2: real;
let
    m1 = if v > u then v else u;
    m2 = if m1 > 0.0 then m1 else 0.0;
    r = m2 -> if pre(r) > m1 then pre(r) else m1;
tel;

node main(x, y : real) returns (P : bool);
var a,b : real;
let
    a = f(x,y);
    b = f(y,x);
    P = a >= x and a >= y and b >= x and b >= y;
    --%PROPERTY P;
tel;
```

If we are interesting in determining which guarantees of the contract `fSpec` of `f` are needed to prove `P`, we should run this command:

```
kind2 <lustre_file> --ivc true --ivc_category contracts --ivc_only_main_node false --composit
```

- `--ivc_category contracts`: because we are only interested in minimizing the contract `fSpec`
- `--ivc_only_main_node false`: because `fSpec` is not the contract of the main node, so we need to consider all nodes
- `--compositional true`: as we want to minimize the contract of `f` and not its implementation, we need to enable compositional analysis

We obtain the following inductive validity core:

```
IVC (2 elements):
  Node f
    Guarantee fSpec[l11c12].guarantee[l6c4][3] at position [l6c4]
    Guarantee fSpec[l11c12].guarantee[l7c4][4] at position [l7c4]
```

30.3 Minimizing over a subset of the assumptions/ guarantees

If you are interested in computing an IVC among a subset of the assumptions or guarantees, you can use the category `annotations`. The assumptions and guarantees that should be considered must be preceded by the keyword `weakly`. All the other assumptions and guarantees will be considered as always present when computing the IVCs.

For instance, we can modify the previous example as follows:

```
contract fSpec(u,v: real) returns(r: real);
let
    weakly guarantee r >= 0.0;
    guarantee true -> r >= pre(r);
    weakly guarantee r >= u;
    guarantee r >= v;
tel;
```

```
kind2 <lustre_file> --ivc true --ivc_category annotations --ivc_only_main_node false --compos
```

We obtain the following inductive validity core:

```
IVC (1 elements):
  Node f
    Guarantee fSpec[l11c12].weakly_guarantee[l6c4][3] at position [l6c4]
```

30.4 Computing all Inductive Validity Cores

If we want to compute ALL the minimal inductive validity cores, we can use the following flags:

```
kind2 <lustre_file> --ivc true --ivc_all true
```

- `--ivc_all true`: specify that we want to compute all the IVCs

31 Minimal Cut Set

The minimal cut set generation is a special mode where Kind 2 computes a minimal subset of the model elements (assumptions, guarantees, stateful equations, or node calls) whose no satisfaction leads to the violation of a property.

To enable minimal cut set generation, run

```
kind2 <lustre_file> --enable MCS
```

31.1 Options

- `--mcs_category {annotations|node_calls|contracts|equations|assertions}` (default: annotations) – Consider only a specific category of elements, repeat option to consider multiple categories
- `--mcs_only_main_node <bool>` (default `false`) – Only elements of the main node are considered in the computation
- `--mcs_all <bool>` (default `false`) – Specify whether all the Minimal Cut Sets must be computed or just one
- `--mcs_approximate <bool>` (default `true`) – Compute a MCS which is minimal with respect to all the counterexamples of the same length that the first counterexample found. This solution can be considered an approximation of a *global* one. Ignored if `--mcs_all` is `true`
- `--mcs_max_cardinality <int>` (default `-1`) – Only search for MCSs of cardinality lower or equal to this parameter. If `-1`, all MCSs will be considered
- `--mcs_per_property <bool>` (default `true`) – If true, MCSs will be computed for each property separately
- `--print_mcs <bool>` (default `true`) – Print the minimal cut set computed
- `--print_mcs_complement <bool>` (default `false`) – Print the complement of the minimal cut set computed (this is equivalent to computing a Maximal Unsafe Abstraction)
- `--print_mcs_legacy <bool>` (default `false`) – Print the minimal cut set using the legacy format
- `--print_mcs_counterexample <bool>` (default `false`) – Print a counterexample for each MCS found (ignored if `--print_mcs_legacy` is true)

- `--mcs_per_property <bool>` (default `true`) – If true, MCSs will be computed for each property separately

31.2 Example

Let's consider the following Lustre code:

```
contract spec(x,y: real) returns(z: real);
let
    weakly assume x = -y;
    weakly assume x >= 0.0;
tel;

node main(x, y : real) returns (z : real);
con
    import spec(x,y) returns (z) ;
noc
  var P : bool;
let
    z = x + y;
    P = z = 0.0;
    --%MAIN;
    --%PROPERTY P;
tel;
```

If you are interesting in determining a minimal set of the weak assumptions of the contract `fSpec` whose no satisfaction leads to the violation of P, you can run this command:

```
kind2 <lustre_file> --enable MCS --mcs_category annotations
```

Note that `--mcs_category annotations` is not required since it is the default value.

The following minimal cut set is printed:

```
MCS (1 elements) for property P:
  Node main
    Assumption spec[l9c12].weakly_assume[l4c4][1] at position [l4c4]
```

In the example above, the `weakly` keywork is used to annotate the assumptions and guarantees to consider for the MCS computation (Kind2 will only try to remove these assumptions and guarantees, all the others will be kept).

Alternatively, if we want to compute a MCS over all the assumptions and guarantees, we can change the category to `contracts`:

```
kind2 <lustre_file> --enable MCS --mcs_category contracts
```

32 Contract Check

Kind 2 provides an option to check the realizability of contracts and refinement types. When an input model includes imported nodes or refinement types, it is particularly important to verify that their associated contracts are realizable, i.e., a component can be constructed such that, for any input satisfying the contract assumptions, there exists some output value that the component can produce to meet the contract guarantees.

To check the contracts of nodes and functions, run:

```
kind2 --enable CONTRACTCK <lustre_file>
```

You can specify a particular node or function to analyze using `--lus_main <node_name>`, a specific refinement type using `--lus_main_type <type_name>`, or a specific constant using `--lus_main_const <const_name>`.

If Kind 2 is able to prove some contract *unrealizable* and the `--print_deadlock` flag is true, Kind 2 will show a deadlocking trace such that all states except the last one satisfy the contract constraints. If the trace only has one state, the state shows input values such that no initial state values satisfy the contract constraints (including the state values chosen as sample). For traces with more than one state, the trace is such that no next state values satisfy the contract constraints from the second-to-last state giving the input values of the last state. Kind 2 will also show a set of conflicting constraints for the last state in the trace.

When the `--check_contract_is_sat` flag is true, Kind 2 will also check whether the unrealizable contract is at least satisfiable, i.e., it is possible to construct a component such that for *at least* one input sequence allowed by the contract assumptions, there is some output value that the component can produce that satisfies the contract guarantees.

In addition, Kind 2 will check the realizability of the component's environment. This check is also important for the top-level contract, as an unrealizable environment specification can lead to the same flawed compositional proof arguments as an unrealizable leaf-level component contract. You can disable this check by passing `--check_environment false`.

33 Assumption Generation

In the early stages of model development and analysis, properties of system components often fail to hold because the assumptions made about a component's environment are insufficient to guarantee these properties, even when component's behavior is correctly specified. When this happens, the system designer must study the counterexample provided by Kind 2, pinpoint the cause, and identify possible restrictions on the environment that the properties need in order to hold —which were perhaps assumed by the designer but were not made explicit.

For instance, consider the following Lustre program:

```
node Arbiter (s1,s2: bool; e1,e2:int) returns(o: int);
con
  guarantee "G1" s1 => o=e1;
  guarantee "G2" s2 => o=e2;
noc
let
  if s1 and s2 then
    o = any { o:int | o = e1 or o = e2 };
  elsif s1 then
    o = e1;
  else
    o = e2;
  fi
tel
```

If we run Kind 2 to check guarantees `G1` and `G2`, Kind 2 determines the properties are invalid, providing a counterexample for each of them. In the first counterexample, `s1` and `s2` are true initially, and the output is equal to the value of `e2`, which falsifies `G1`. In the second counterexample, `s1` and `s2` are true initially, and the output is equal to the value of `e1`, which falsifies `G2`.

From the description of the counterexamples, it is evident that one potential missing assumption is that `s1` and `s2` are never simultaneously true. If that is indeed the case, explicitly stating this assumption in the contract of the node, allows Kind 2 to prove both properties valid.

Generating the missing assumptions for straightforward examples like the one above is not very difficult. However, in more realistic scenarios, this task can become quite challenging. To help system designers in those situations, Kind 2 offers a post-analysis that can be enabled by passing the command-line option `--assumption_gen true`. When this option is enabled, Kind 2 will automatically generate assumptions that are strong enough to prove the set of falsified properties in the verification analysis while not being overly restrictive. Note that just finding sufficient assumptions is not challenging. The challenge is to find minimally restrictive assumptions which can then be recognized as realistic by the designer.

For the example above, Kind 2 generates the assumption: `(not s2) or (not s1) or (e1 = e2)`. Notice that this assumption is a weaker version of the one mentioned above, as it considers a third case where the inputs `e1` and `e2` equal.

The functionality generates one-state constraints on a node’s environment, but it currently lacks the capability to generate two-state properties, which presents a significantly more complex challenge.

34 A Lucent Primer

This page is a self-contained introduction to Lucent, the input language of Kind 2, intended for new users. Lucent is an extension of Lustre (tutorial, reference); we assume readers have no prior knowledge of Lustre.

34.1 Basic Concepts

34.1.1 Lucent Nodes

Lucent is a language for modeling and implementing reactive systems in the synchronous model. It can be seen indifferently as either a declarative parallel programming language or as an executable specification language. The most basic unit of computation in a Lucent program, or model, is a **node**, which can be viewed as a stream transformer: it takes streams of input and produces streams of output. Operationally, a node reads its input and generates its output incrementally in discrete *timesteps*, or cycles, determined by an abstract global clock. At each cycle, all output values are assumed to be computed instantaneously from the current input and state values. By default, all nodes in a model compute synchronously and in parallel according to the global clock.

A **stream** is an infinite sequence of values, all of the same (given) type. Hence, a Lucent node can be viewed as modeling an infinite sequence of discrete timesteps, where at each timestep, each node variable takes its next value.

Below, the node `Combine` takes as input two integer streams x and y, and produces integer stream z as output. If we consider $x = (x_0, x_1, \dots)$ and $y = (y_0, y_1, \dots)$, then `Combine` produces output $z = (x_0 + 2 \cdot y_0,\ x_1 + 2 \cdot y_1,\ \dots)$ (or more concisely, $z_n = x_n + 2 \cdot y_n$ at each timestep n).

It is not possible to specify a stream pointwise in Lucent, so when we write , say, we are writing a mathematical statement about stream , not an equation in Lucent.

Notice that `z = x + 2*y` is an equation between streams of integers. The operators `=`, `+` and `*` are stream operators obtained by lifting to streams the corresponding

operators over integers. The same is true of concrete constants in Lucent, such as `2` below, which are streams with the same value at each time step. Lucent respects typical rules of operator precedence, so `x + 2*y` will be parsed as `x + (2*y)` rather than `(x + 2)*y`.

```
node Combine(x: int; y: int) returns (z: int);
let
  z = x + 2*y;
tel
```

The first line of `Combine` is referred to as the **node interface**, where the node's inputs and outputs, and their types, are declared.

The code block surrounded by `let` and `tel` denotes the **node implementation** (or **node body**), where the node's outputs are defined in terms of the node's inputs. A node implementation is composed of a set of equations of the form `<var> = <expr>`, where `<var>` is an output variable or a local variable (see below) and `<expr>` is an expression in terms of any of the variables that are in scope.

Nodes can have more than one output stream as exemplified by the node `TwoOuts` below.

```
node TwoOuts(x: int) returns (double: int; square: int);
let
  double = x + x;
  square = x * x;
tel
```

Another optional component that can be added to a Lucent node is a set of **local declarations**. The local variables and constants declared in this section can be used in the node implementation, but they are not exposed in the node interface.

Finally, **global constants** can be declared outside of the node body, and are visible within every node.

Below is another version of `Combine`, where the value `2` is stored in a global constant `C` and the local variable `l` is used to store an intermediate computation.

```
const C: int = 2;
node Combine(x: int; y: int) returns (z: int);
var l: int;
```

```
let
  l = C*y;
  z = x + l;
tel
```

The order of the equations in the body of a node is immaterial. However, the definition of a variable provided by the equations cannot be *circular*, as explained in Declarative Semantics.

In Lucent, identifiers (for constants, variables, types, and keywords) are delimited by whitespace characters, separators such as parentheses and semicolons, and other symbols such as `+`, `*` and so on, as in most programming languages. Whitespace is, however, not semantically meaningful. For instance, indentation does not change the parsing of an expression.

34.1.2 Node Analyses

Lucent was designed to be a programming language. Well-formed Lucent nodes are executable in the sense that they can be compiled to executable programs computing their output values incrementally from their input values and internal state.

Here, we are mostly interested in *analyzing* Lucent programs and their possible behavior with a tool like Kind 2.

A basic form of analysis that can be applied to a Lucent program is **node simulation**. During simulation, the user specifies a number n of timesteps to simulate, as well as the first n values of each input variable. Given this information, the first n values of each output variable are computed. For the `Combine` node above, if the user performed simulation with $n = 3$ and with given input stream prefixes $x = (1, 2, 3)$ and $y = (4, 5, 6)$, the output value $z = (9, 12, 15)$ would be computed.

Another form of analysis is **property checking**, where the user specifies a property in the node body (in the form of a Boolean expression) to be proven or disproven **invariant**, that is, true at every time step. For example, the conditional property `y > 0 => l > y` in the node below would be proven invariant. In contrast, the property `z > 0` would be disproven in the `Combine` node, as `z` is negative in timesteps where both `x` and `y` are negative.

```
const C: int = 2;
(* Example with
   two properties
```

```
*)
node Combine(x: int; y: int) returns (z: int);
var l: int;
let
  l = C*y;
  z = x + l;

  check y > 0 => l > y; -- invariant
  check z > 0;          -- not invariant
tel
```

We discuss properties in greater detail in Specifying and Checking Properties below.

34.2 Comments

The example above shows the two ways to add comments in Lucent programs. Single line comments are introduced by the character sequence `--`. Multiline comments are delimited by the sequences `(*` and `*)`. Nested multiline comments are not allowed.

34.3 Primitive Types

Lucent's primitive types are `bool`, `int`, and `real`. Informally, we say that `bool` is the type of Boolean values (`true`, `false`). Strictly speaking, `bool` is the type of *streams* of Boolean values. We identify the two for brevity since there is no possibility of confusion as all values in Lucent are streams. The same is true for the other types.

It is not possible to refer directly to the scalar values in a stream in Lucent. Even constants, such as `true`, `2`, `3.6` denote streams of values, not individual values.

In the **idealized** semantics of Lucent, `int` is the type of mathematical (infinite precision) integers, and `real` is the type of real numbers. Lucent compilers approximate that semantics by using machine integers for `int` and floating point numbers for `real`. In contrast, Kind 2 is faithful to the idealized semantics.

Lucent supports the Boolean operators `not`, `and`, `or`, `xor`, and `=>` (implies), as well as the arithmetic operators `+`, `-` (both unary and binary), `*`, `/`, `mod`, and `div` (integer division), all with the expected arity and (pointwise) semantics. Integer division and modulo follow the Euclidean definition, as in SMT-LIB: `m mod n` is never negative, and `m div n` is rounded accordingly (towards negative infinity when `n` is

positive), so `(-7) div 2` is `-4` and `(-7) mod 2` is `1`, where C gives `-3` and `-1` (see Integer division and modulo). The arithmetic operators (`+` and so on) are overloaded as they apply both to `int` and `real` terms. The binary operators, however, are applicable only to arguments of the same type (both `int` or both `real`). Numerals (`0`, `1`, …) have type `int` while decimals (e.g., `0.0`, `31.97`) have type `real`.

Alongside the mathematical `int` and `real`, Lucent offers fixed-width **machine integers**, signed as `sint<N>` and unsigned as `uint<N>` for a width `N` in bits, with the concise names `int8`, `uint8`, `int16`, and so on for the widths 8, 16, 32, and 64. Unlike `int`, these types are finite: their values are built by applying a conversion operator to a literal, as in `uint8 27`, and their arithmetic wraps around modulo the width of the type, as in C, with signed values represented in two's complement. They also support the bitwise operators `&&`, `||`, and `!`, and the shifts `lsh` and `rsh`. See Machine Integers for the conversion rules between widths and the solver restrictions these types imply.

34.4 Conditionals

Equations can be specified conditionally with `if` *statements*, which are closed with `fi`:

```
node Sign(x: int) returns (neg, pos: bool);
let
  if x < 0 then
    neg = true;
    pos = false;
  elsif x > 0 then
    neg = false;
    pos = true;
  else
    neg = false;
    pos = false;
  fi
tel
```

Each condition is evaluated once per timestep, and every variable takes its value according to its definition in the equation in the active branch.

Conditions can also be encoded as `if` *expressions*:

```
if <expr_0> then <expr_1> else <expr_2>
```

where `<expr_0>` has type `bool`, and `<expr_1>` and `<expr_2>` must have the same type.

```
node Max(x, y: int) returns (m: int);
let
  m = if x > y then x else y;
tel
```

34.5 Temporal Operators

Lucent contains two temporal operators: the binary operator `->` (pronounced “arrow” and not to be confused with `=>`) and the unary operator `pre`.

The arrow operator is an *initialization* operator, where the expression `a -> b` denotes the stream whose first value is equal to the first value of stream `a`, and whose nth value is equal to the nth value of stream `b` for every $n > 0$. For example, if $\mathtt{a} = (-1, -1, -1, \dots)$ and $\mathtt{b} = (1, 2, 3, \dots)$, then $\mathtt{a\ ->\ b} = (-1, 2, 3, \dots)$.

The `pre` operator can be viewed as referencing the previous value at every timestep—the expression `pre a` denotes the stream whose value at step n is equal to the value of stream `a` at step $n - 1$. For example, if $\mathtt{b} = (1, 2, 3, \dots)$, then $\mathtt{pre\ b} = (?, 1, 2, \dots)$. Notice that with these semantics, `pre b` is undefined in the initial timestep (denoted by the question mark here).

Kind 2 treats undefined expressions as **underspecified**. That is, when simulating the stream `pre b`, it could take values $(-23, 1, 2, \dots)$, $(79, 1, 2, \dots)$, etc. In other words, Kind 2 assigns the first element of `pre b` an arbitrary integer. Consistently with that, a property of a node containing `pre`s is considered invariant only if it holds at every step, regardless of the value assigned to the first element of any stream resulting from a `pre` application.

Because `pre` creates underspecified streams, we can combine it with `->` to obtain fully specified streams. For example, if $\mathtt{b} = (1, 2, 3, \dots)$, then $\mathtt{0\ ->\ pre\ b} = (0, 1, 2, 3, \dots)$, where the arrow operator supplies the initial value 0 for the resulting stream. If an application of `pre` occurs without a corresponding application of `->`, the `pre` is **unguarded**. While unguarded `pre`s are allowed in Lucent, Kind 2 will

produce warnings for nodes that contain them as this is usually an oversight by the user and may lead to unexpected results.

The `pre` operator has the same precedence as other unary operators such as `not`. For example, `pre x + y` is read as `(pre x) + y`, not as `pre (x + y)`. Note that `pre` distributes over all non-temporal operators. For instance, the expression `pre (x + y)` is equivalent to `pre x + pre y`.

To further reinforce how operators work over streams, the computation of the expression `1 -> (1 + pre x)` is illustrated in the table below.

Expression	0	1	2	**...**	n
`1`	1	1	1	...	1
`x`	x_0	x_1	x_2	...	x_n
`pre x`	$?$	x_0	x_1	...	x_{n-1}
`1 + pre x`	$1+?$	$1 + x_0$	$1 + x_1$	...	$1 + x_{n-1}$
`1 -> (1 + pre x)`	1	$1 + x_0$	$1 + x_1$	...	$1 + x_{n-1}$

Using temporal operators, we can define a `Counter` node as follows.

```
node Counter(init: int) returns (out: int);
let
  out = init -> pre out + 1;
tel
```

In `Counter`, the output stream `out` is initialized to the input initialization value `init`, and it is incremented at every timestep. Notice that `out` is recursively defined —the $n+1$st value of `out` is equal to the nth value of `out` plus 1, except in the *base case* of initialization.

The `pre` and `->` operators provide a declarative and mathematically elegant way to define **stateful** computations. An alternative, operational way to understand the functionality of node `Counter` is that `init` is an input variable and `out` is a *state* variable. Initially, the value of `out` is that of `init`. At each successive iteration, the new value of `out` is its old value (denoted as `pre out`) plus one.

A deceptively difficult example is defining in Lucent a stream with value $(1, 2, 3, 3, 3, \dots)$, with infinite repetitions of 3 from the third step on. A first guess

might be the term `1 -> (2 -> 3)` or perhaps the term `(1 -> 2) -> 3`. However, both of these streams will omit the value 2, as they take the initial value from the first argument of the outer arrow (which is `1` in both cases) and the non-initial values from the second argument of the outer arrow (which is a stream of `3`s in both cases). A key insight is that the `pre` operator can also be viewed as a *right-shift operator* on streams. From this, the correct answer is `1 -> pre (2 -> 3)`, which takes the initial value 1 and the remaining values from the stream $(?, 2, 3, 3, 3, \dots)$.

The table below helps illustrate the difference between the various expressions above.

Expression	0	1	2	3	**...**
`1`	1	1	1	1	...
`2`	2	2	2	2	...
`3`	3	3	3	3	...
`1 -> 2`	1	2	2	2	...
`2 -> 3`	2	3	3	3	...
`pre (2 -> 3)`	$?$	2	3	3	...
`1 -> (2 -> 3)`	1	3	3	3	...
`(1 -> 2) -> 3`	1	3	3	3	...
`1 -> pre (2 -> 3)`	1	2	3	3	...

A node that generates the stream $(1, 2, 3, 3, 3, \dots)$ from no inputs can then be defined as follows.

```
node N() returns(y: int);
let
   -- defining output stream (1, 2, 3, 3, 3, ...)
   y = 1 -> pre (2 -> 3);
tel
```

Another deceptively difficult example is the following Lucent node which outputs the stream of all Fibonacci numbers in increasing order. Because `Fib` is defined in terms of the two previous Fibonacci values, the first *two* steps need to be initialized. The example is tricky and may require some thought for those new to Lucent.

```
node Fibonacci() returns(Fib: int);
let
  Fib = 1 -> pre (1 -> Fib + pre Fib);
tel
```

The example can perhaps be easier to see by introducing local names for the subexpressions on the equation’s right-hand side.

```
node Fibonacci() returns(Fib: int);
  var preFib: int;
  var prepreFib: int;
let
  preFib = 0 -> pre Fib;
  prepreFib = 1 -> pre preFib;
  Fib = preFib + prepreFib;
tel
```

34.6 Frame Blocks

A `frame` block names a set of variables, gives them optional initial values, and defines them in a body between `let` and `tel`. Any variable the body leaves undefined at a step *stutters*: it keeps the value it had at the previous step, starting from its initialization.

```
node Hold(latch: bool; v: int) returns (out: int);
let
  frame ( out )
  out = 0;
  let
    if latch then
      out = v;
    fi
  tel
tel
```

Here `out` starts at `0`, takes the value of `v` whenever `latch` is true, and otherwise holds its previous value.

34.7 Declarative Semantics

Lucent has a **declarative** semantics, meaning that the order of equations in node bodies does not matter. Because of this, node equations should not be viewed imperatively as assignments; instead, a node body is a set of stream constraints of the form `<var> = <expr>`.

To illustrate this concept, consider the following `Factorial` node which outputs a stream of factorial numbers (the nth value of the stream is $n!$). When defining output stream `F`, we can reference the helper stream `N` before it is defined.

```
node Factorial() returns (F: int);
var N: int;
let
  -- all the factorial numbers
  F = 1 ->  N * (pre F);
  -- all the natural numbers
  N = 0 -> (pre N) + 1;
tel
```

Even though Lucent has a declarative semantics and allows recursive definitions, circular definitions are rejected. For example, the following node is invalid Lucent because the nth value of `out1` is defined in terms of the nth value of `out2`, and the nth value of `out2` is defined in terms of the nth value of `out1`.

```
node Circular() returns (out1, out2: int);
let
  out1 = out2 + 1;
  out2 = out1 - 1;
tel
```

In fact, there are no values for the streams `out1` and `out2` that satisfy both equations. However, even if it is possible to satisfy all equations, as in the following example, any node with a circular dependence is conservatively rejected.

```
node Circular() returns (out1, out2: int);
let
  out1 = out2;
```

```
  out2 = out1;
tel
```

Note that there is no circularity in the definition of local variable `N` of node `Factorial` since `N` is defined in terms of `pre N`, and not in terms of `N` itself.

34.8 Composite Types

In addition to the primitive types, Lucent provides records, arrays, tuples, sets, maps, and algebraic datatypes.

34.8.1 Records

Record types have the syntax

```
struct { <field_1>: <type_1>; ...; <field_n>: <type_n> }
```

They must be named and declared with a global **type declaration** of the form

```
type <ty_name> = <type>;
```

Record values can be constructed with the syntax

```
<ty_name> { <field_1> = <expr_1>; ...; <field_n> = <expr_n> }
```

and destructed with the syntax

```
<record_term>.<field>
```

as seen in the next example.

```
type sensorData = struct { speed: real; height: real; direction: int };

node AdjustSensorData(data: sensorData) returns (out: sensorData);
  var h: real;
let
  h = if data.height < 0.0 then 0.0 else data.height;
  out = sensorData { speed = data.speed;
                     height = h;
```

```
                    direction = data.direction };
tel
```

See Records for the remaining record operations, such as element updates.

34.8.2 Arrays

Array types have the syntax

```
<element_type>^<numeral>
```

Values of an array type can be constructed in two different ways. Lucent supports the **array literal** syntax of the form

```
[<element_1>, ..., <element_n>]
```

as well as the (constant) **array constructor** syntax of the form

```
<element>^<length>
```

Array elements can be accessed with the standard **array access** syntax `<array_var>[<index>]`, with zero-based indexing.

```
node TwoArrays() returns (out1: bool^5; out2: int^4);
let
  out1 = [true, true, false, true, false];
  out2 = 1^4;   -- equivalent to out2 = [1, 1, 1, 1]
tel
```

```
node Nth(a: int^10; k: int) returns (out: int);
let
  out = if 0 <= k and k < 10 then a[k] else a[0];
tel
```

See Arrays for element update, structural equality, and inductively defined arrays.

34.8.3 Tuples

A tuple type is written `[<type_1>, ..., <type_n>]`. Tuples are constructed with the syntax `'(<expr_1>, ..., <expr_n>)`, and their components are read back by

position with `<tuple>[<index>]`, using zero-based indexing. The index must be a concrete numeral, since it selects a component rather than computing one.

```
type Pair = [int, bool];

node Swap(p: Pair) returns (q: [bool, int]);
let
  q = '(p[1], p[0]);
tel
```

34.8.4 Sets and Maps

The type `set<T>` denotes streams of finite sets of elements of type `T`, and the type `map<K, V>` denotes streams of finite maps from keys of type `K` to values of type `V`. Set literals are written with braces, and map literals with the `map[...]` constructor:

```
node N(s: set<int>; m: map<int, int>) returns (u: set<int>; v: int);
let
  u = s + { 1, 2, 3 };
  v = m[0];
tel
```

The set operators are union `+`, intersection `*`, difference `-`, and membership `in`. A map is updated (functionally, producing a copy) with `m[k := v]`, read with `m[k]`, and restricted with `m - s`, which removes every key in the set `s`. Membership `k in m` tests for a key.

Empty map and set literals carry no element type of their own, so the user must annotate types: `{}@<int>` and `map[]@<int, int>`.

Reading a key that a map does not bind, as in `m[k]` where `k` is absent, yields an unconstrained value rather than an error. The read is still *functional*, though: the same key always yields the same value at the same timestep.

See Sets and Maps for the full operator set and the restrictions on element and key types.

34.8.5 Algebraic Datatypes

An algebraic datatype packages a fixed set of *constructors*, each carrying zero or more named fields. They are introduced with the `datatype` keyword, with constructors separated by `|`:

```
datatype Shape =
  | Circle (radius: real)
  | Rectangle (width: real, height: real)
  | Point;
```

A value is built by applying a constructor to its field values, with nullary constructors such as `Point` written without parentheses. Values are taken apart with a `match` expression, whose arms must cover every constructor. Each arm names fresh variables for the fields of its constructor, in scope only within that arm:

```
datatype Shape = Circle (radius: real) | Rectangle (width: real, height: real);

node Area(s: Shape) returns (a: real);
let
  a = match s with
    | Circle (r)       : 3.14 * r * r
    | Rectangle (w, h) : w * h
  end;
tel
```

Datatypes may be recursive, which makes it possible to describe unbounded structures such as lists:

```
datatype IntList = Cons (head: int, tail: IntList) | Nil;
```

See Algebraic Datatypes for testers, selectors, and polymorphic datatypes.

34.9 Enumerations and Subranges

Two named types describe restricted sets of scalar values.

An **enumeration** is a finite set of named constants:

```
type Color = enum { Red, Green, Blue };
```

A **subrange** describes the integers within given inclusive bounds, with the syntax `subrange [LB, UB] of int`. Either bound may be `*`, leaving that side unbounded, and either may be a symbolic constant expression rather than a literal:

```
type Percent = subrange [0, 100] of int;
type Pos = subrange [1, *] of int;
```

A subrange on an input or a free constant is an *assumption* Kind 2 may rely on; a subrange on an output, a local variable, or a defined constant is a *proof obligation* Kind 2 must discharge. The node below type-checks, but Kind 2 falsifies the obligation on its output, since nothing prevents `x + y` from exceeding `100`:

```
type Percent = subrange [0, 100] of int;

node Add(x, y: Percent) returns (z: Percent);
let
  z = x + y;
tel
```

Kind 2 reports this as a failed property named for the position of the offending declaration, alongside a counterexample — here, any two inputs summing above `100`.

Subranges can be viewed as special cases of refinement types, which are discussed next.

34.10 Refinement Types

A **refinement type** restricts a base type with a predicate. It is written `subtype { <var>: <base_type> | <predicate> }`:

```
type Nat = subtype { x: int | x >= 0 };
```

The base type can be any type, including another refinement type, and refinement types may appear inside composite types (as the element type of an array or set, for instance).

Where a variable is declared, a more concise form is available: writing `<var>: <base_type> | <predicate>` in a node's interface or local declarations means the same thing.

```
node Sqrt(x: real | x >= 0.0) returns (y: real | y >= 0.0);
```

Refinement types follow exactly the rule given for subranges above: a refinement type on an input or a free constant is an **assumption**, while one on an output, a local variable, or a defined constant is a **proof obligation**. So in `Sqrt`, Kind 2 may assume `x >= 0.0` and must prove `y >= 0.0`.

See Refinement Types for the treatment of defined versus free constants, and for refinement types nested inside structured types.

34.11 Abstract Types

An **abstract type** is a type declared without a definition:

```
type T;
```

Kind 2 treats it as an uninterpreted domain: the only operations on its values are equality `=` and disequality `<>`, and nothing is assumed about how many values it holds. This is useful to model data whose representation is irrelevant to the properties being checked.

```
type T;

function IdT(x: T) returns (y: T);
let
  y = x;
tel
```

34.12 Composition

A Lucent model can be hierarchically defined by defining nodes in terms of other nodes through the use of **node applications**. Revisiting the `Counter` node, we can use node applications to instantiate two distinct counter streams. In the following example, the output streams `ctr1` and `ctr2` of node `Top` are defined using

expressions that contain node applications. More specifically, output variable `ctr1` is defined as the stream output by node `Counter` when applied to input `0`, incremented by `3`, and the output variable `ctr2` is defined as the stream output by node `Counter` when applied to input `5`. Output `P1` is a Boolean stream representing the property that `ctr2` is greater than `ctr1`.

Note that nodes can have no inputs (as node `Top` below) or no outputs.

```
node Top() returns (ctr1, ctr2: int; P1: bool);
let
  ctr1 = Counter(0) + 3;
  ctr2 = Counter(5);
  P1 = (ctr2 > ctr1);
tel

node Counter(init: int) returns (out: int);
let
  out = init -> pre out + 1;
tel
```

Node applications must respect the expected type checking rules: each argument of the application of a node N, which can be any stream-denoting expression, must have a type that matches the type of the corresponding input parameter in N's interface. Similarly, the return type of N must be a valid type for the expression that contains the node application. For example, the return type of `Counter` matches the expected type for the first argument of the `+` operator in the expression `Counter(0) + 3`.

Note that the definition of node `Top` includes an application of node `Counter`, even though `Top` is defined before `Counter`. Similarly to equations in a node body, the order of node definitions in a Lucent model is immaterial. However, the application graph cannot contain cycles. In other words, a node cannot be defined, directly or indirectly (through subnodes), in terms of itself.

In general, an application of a node with a single output stream of some type T can occur anywhere an expression of type T can occur on the right-hand side of an equation in a node's body. In contrast, an application of a node with multiple outputs can occur only in an equation of the form

```
(<var_1>, ..., <var_n>) = <node_name>(<arg_1>, ..., <arg_m>);
```

or

```
<var_1>, ..., <var_n> = <node_name>(<arg_1>, ..., <arg_m>);
```

where `<var_1>`, …, `<var_n>` are local or output variables of the node containing the application, with types matching the types of the outputs of the applied node `<node_name>`, in the same order as in that node's interface.

```
node Top(x: int) returns (P1: bool);
  var positive: bool;
  var nonnegative: bool;
let
  (positive, nonnegative) = N(x);
  P1 = positive => nonnegative;
tel

node N(x: int) returns (positive, nonnegative: bool);
let
  positive = (x > 0);
  nonnegative = (x >= 0);
tel
```

34.12.1 Functions

Besides `node`, the language provides the keyword `function`, used in exactly the same way but with stricter semantics: a function's outputs must be a *non-temporal* combination of its inputs (i.e., *combinational*). A function may not use `->`, `pre`, `fby` or `restart`, and it may only call other functions (not nodes). Functions are, in other words, stateless.

```
function Abs(x: real) returns (y: real);
let
  y = if x < 0.0 then -x else x;
tel
```

A function behaves as a mathematical function: the same inputs always yield the same outputs, whatever the timestep.

34.12.2 Imported Nodes

A node or function declared `imported` has an interface but no body:

```
node imported Sensor(t: int) returns (reading: real);
```

For example, this is useful to model an external routine whose specification is known but whose implementation is not. For Kind 2 it means the component is *always* abstract: it is represented solely by its contract (see Contracts below), and Kind 2 never looks inside it, because there is nothing to look at. With no contract, the implicit one is `assume true; guarantee true;`, which says nothing at all — so an imported node without a contract may produce any (well-typed) output whatsoever.

34.12.3 Polymorphic Nodes

Nodes and functions may take type parameters, declared in angle brackets after the name:

```
node SafePre<T>(x: T) returns (y: T);
let
  y = x -> pre x;
tel

node Top() returns (y1: int; y2: bool);
let
  y1 = SafePre@<int>(0);
  y2 = SafePre@<bool>(false);
tel
```

The `@<...>` instantiation at the call site is usually optional: Kind 2 infers the type arguments *bottom-up*, by matching the declared parameter types against the types of the actual arguments, so `SafePre(0)` (without the annotation) would also work here.

However, in some cases, Kind 2 cannot infer the type bottom-up.

```
node Default<T>() returns (y: T);
let
  y = any@<T>;
tel

node Top() returns (b: bool);
```

```
let
  b = Default();
tel
```

Kind 2 rejects this during type checking, with

```
Call requires explicit annotation; type variable T cannot be inferred bottom-up
```

The fix is to supply the type argument explicitly (here, writing `Default@<bool>()`). In short: leave the annotation out, and add one at the call site if Kind 2 asks for it.

Type declarations can take parameters in the same way, so a user-defined type can be polymorphic as well. Such a type acts as a *type constructor*: applying it to types yields a type.

```
type Pair<T; U> = [T, U];

node Swap<T; U>(x: Pair<T; U>) returns (y: Pair<U; T>);
let
  y = '(x[1], x[0]);
tel
```

34.13 Lazy Operators

Most operators are **eager**: they evaluate every operand, whichever one the result turns out to depend on. Both branches of `if <cond> then <e1> else <e2>` are evaluated at each step, and so are both operands of `and`, `or`, and `=>`.

Each has a **lazy** counterpart that evaluates an operand only when the result depends on it. For example:

Eager	Lazy	`e2` evaluated
`if c then e1 else e2`	`when c then e1 else e2`	only when `c` is false
`e1 and e2`	`e1 and then e2`	only when `e1` is true
`e1 or e2`	`e1 or else e2`	only when `e1` is false
`e1 => e2`	`e1 ==> e2`	only when `e1` is true

The main motivation for lazy operators is not efficiency. Consider reading a field of an algebraic datatype: the selector `x.val` carries a proof obligation that `x` was built with the constructor that has a `val` field.

```
datatype Option = None | Some (val: int);

node Unwrap(x: Option) returns (y: int);
let
  y = when Some?(x) then x.val else 0;
tel
```

Written with `if ... then ... else ...` instead, the same node is falsified: both branches are evaluated at every step, so `x.val` is read even when `x` is `None`, and the selector's obligation fails. Under `when`, the `then` branch is evaluated only where `Some?(x)` holds, and the obligation is discharged.

The lazy Boolean operators can be used to guard selectors in the same way, as in `Some?(x) ==> x.val > 0`, as well as the division in `x <> 0 and then y / x > 1`.

The same laziness is available at statement level, in the `when` and `cond` blocks described next.

34.13.1 Lazy Blocks

The lazy counterpart of the `if` statement is the `when` block, closed with `end`:

```
when <cond> then
   <equations>
else
   <equations>
end
```

Further branches are written by nesting another `when` block inside the `else` branch. A `cond` block is similar, with any number of guarded branches and an `otherwise` clause:

```
cond
  | <cond_1>:
     <equations>
  | <cond_2>:
```

```
    <equations>
  otherwise:
    <equations>
end
```

In both, only the selected branch is evaluated, with the consequences described above. A branch may contain temporal operators and calls to nodes (see below for what they mean there), but `if` blocks and lazy blocks may not be nested inside one another.

A `match` block is the statement-level counterpart of the `match` expression of the Algebraic Datatypes section, standing to it as the `if` statement stands to the `if` expression. Its arms hold equations rather than a value, which is what makes it worth having: an arm can define several variables at once.

```
datatype Shape = Circle (radius: real) | Rectangle (width: real, height: real);

node Area(s: Shape) returns (a: real; round: bool);
let
  match s with
  | Circle (r):
    a = 3.14 * r * r;
    round = true;
  | Rectangle (w, h):
    a = w * h;
    round = false;
  end
tel
```

Only the selected arm is evaluated, so the fields an arm names are guarded just as `Some?(x)` guards `x.val` above. Every variable defined in one arm must be defined in all of them, and the arms must cover every constructor, unless the block sits inside a frame block, where a value matched by no arm leaves the variables to stutter.

Laziness also changes what “the previous value” means. Inside a lazy branch, or in the right operand of a lazy Boolean operator, `pre x` refers to the value of `x` the last time *that branch was selected*, which may be several steps earlier. Likewise, `->` selects its left operand the first time the branch is selected, and a node called in the branch only advances at the steps where the branch is selected. When a lazy block sits inside a frame block, `last x` is the dependable way to say *the value at the immediately preceding timestep*.

34.14 Nondeterministic Choice

The operator `any { <var>: <type> | <predicate> }` denotes an arbitrary stream of values of the given type satisfying the predicate:

```
node N(y: int) returns (z: int);
var l: int;
let
  l = any { x: int | x mod 2 = 1 };
  z = y + l;
tel
```

Here `l` is some odd stream, with no further commitment as to which — it may take a different odd value at every step.

The variant `choose { ... }` is similar to `any { ... }`, but functional (that is, it always produces the same value for the same values of the variables occurring in its predicate).

Both operators can be written with an explicit type instantiation instead of a predicate, as in `any@<bool>`, which denotes an arbitrary Boolean stream.

34.15 Type Ascription

The ascription operator `(<expr>: <type>)` checks that an expression satisfies a type. It generates a proof obligation; it does *not* introduce an assumption. So given the type `Nat` above, if `x` is an input of type `int`, then `(x: Nat)` obliges Kind 2 to prove `x >= 0`.

Ascription applies to ordinary types as well, where it acts as a static check rather than a proof obligation: `(1 + 2: bool)` is rejected during type checking.

34.16 Common Auxiliary Nodes

While the temporal operators `->` and `pre` may not seem very powerful, they can be used to define auxiliary temporal operators, presented below.

```
-- Y is true iff X has been true so far
node Sofar ( X : bool ) returns ( Y : bool ) ;
let
 Y = X -> (X and (pre Y)) ;
tel

-- Z is true iff X has been true at some point in the past,
-- and Y has been true since then.
node Since ( X, Y : bool ) returns ( Z : bool ) ;
let
  Z =  X or (Y and (false -> pre Z)) ;
tel

-- Y is true iff X was true in the initial timestep
node Initially(X: bool) returns (Y: bool);
let
  Y = X -> pre Y;
tel

-- Y is true iff X has been true at least once
node Once(X : bool) returns (Y : bool);
let
  Y = (false -> pre Y) or X;
tel
```

34.17 Specifying and Checking Properties

Property checking was introduced informally at the start of this page. This section covers the constructs the language provides for it.

34.17.1 Properties

A property to be proven invariant is written with a `check` statement in the body of a node. Properties may be named, which makes Kind 2's output easier to read when there are several:

```
node Count(trigger: bool) returns (n: int);
let
  n = (if trigger then 1 else 0) + (0 -> pre n);

  check "nonneg" n >= 0;
```

```
  check "small" n <= 10;
tel
```

Kind 2 reports each property as valid or falsified independently. For `Count` above, `"nonneg"` is proven invariant, while `"small"` is falsified, and Kind 2 prints a counterexample: an input sequence, with the resulting values of every stream, that drives the model to a state violating the property.

34.17.2 Choosing What to Analyze

By default, Kind 2 analyzes the *top nodes* of a model: those no other node calls. A node can be designated explicitly by adding the annotation `--%MAIN;` to its body, or by passing `--lus_main <node_name>` on the command line. If any main node is designated, only main nodes are analyzed.

34.17.3 Reachability Properties

Invariants say that something never happens. The dual — that something *can* happen — is written with `check reachable`, which asks Kind 2 to find a witness trace rather than to rule one out:

```
check reachable "can reach ten" n = 10;
```

The search can be bounded: `from <int>` requires the witness to take at least that many steps, `within <int>` at most that many, and `at <int>` exactly that many.

Reachability checks are worth writing even in a model whose invariants all hold, because they catch a model that is accidentally over-constrained—one where the interesting states are simply unreachable.

34.17.4 Conditional Properties

Properties are often of the form “in this situation, this behavior”. Written naively as `check B => A;`, such a property is trivially true whenever the situation `B` never arises, which can hide a modeling error. The language provides a dedicated syntax for this case:

```
check A provided B;
```

This checks that `B => A` is invariant *and*, separately, that `B` is reachable, so a vacuously true property is reported as such.

34.17.5 Quantifiers

Properties and contracts may use the quantifiers `forall` and `exists`. They may appear only in specifications, not in the equations that define a node's outputs, and they are indispensable for models parameterized by a size:

```
check forall (i: int) 0 <= i and i < n => a[i] >= 0;
```

A quantified variable may be given a refinement type with the concise syntax `x: <type> | <predicate>`, so the property above can also be written `forall (i: int | 0 <= i and i < n) a[i] >= 0`.

34.18 Contracts

Properties state facts about one node. A **contract** describes a node's obligations to, and expectations of, its callers, which is what lets Kind 2 analyze a large model *compositionally*: each node is verified once against its own contract, and callers reason about it through that contract instead of re-examining its body.

A contract is a set of **assumptions**, which the caller must establish, and **guarantees**, which the node must then deliver. It is written inline between a node's interface and its body, delimited by `con` and `noc`:

```
node Divide(x, y: real) returns (z: real);
con
  assume "nonzero divisor" y <> 0.0;
  guarantee "exact" z * y = x;
noc
let
  z = x / y;
tel
```

The meaning is: *if the assumptions always hold, the guarantees always hold.* The obligation is shared. Kind 2 checks that `Divide` delivers `"exact"` given `y <> 0.0`, and separately checks, at every call site, that the caller really does supply a nonzero

`y`. An assumption may not refer to the node's outputs at the current step — the caller has no control over those — though it may refer to them under a `pre`.

A contract may also declare **ghost variables** and constants with `var` and `const`. These are visible to the contract but not to the node body:

```
con
  var once: bool = trigger or (false -> pre once);
  guarantee once => count > 0;
noc
```

34.18.1 Modes

Requirements in a specification document are usually of the form "in this situation, behave this way." A **mode** captures that shape directly: it pairs a set of `require` clauses (the situation) with a set of `ensure` clauses (the required reaction).

```
node Times(lhs, rhs: real) returns (res: real);
con
  mode absorbing (
    require lhs = 0.0 or rhs = 0.0;
    ensure res = 0.0;
  );
  mode positive (
    require lhs > 0.0 and rhs > 0.0;
    ensure res > 0.0;
  );
noc
let
  res = lhs * rhs;
tel
```

A mode is equivalent to the guarantee `requires => ensures`, but modes are useful because Kind 2 uses them to report *which* mode was active in a counterexample, and it checks the set of modes for **exhaustiveness**, warning when the modes leave some situation unspecified. A mode can be referred to elsewhere in the contract by name, as in `require not ::absorbing;`.

34.18.2 Contract Nodes

A contract can also be written separately and reused, as a **contract node** — like an ordinary node, but introduced with `contract` and containing only contract items. It is

brought into a node's contract with `import`, which merges the imported assumptions, guarantees, and modes into the importing contract:

```
contract Spec(x: real) returns (y: real);
let
  assume x >= 0.0;
  guarantee y >= 0.0;
tel

node Sqrt(x: real) returns (y: real);
con
  import Spec(x) returns (y);
noc
let
  y = if x <= 0.0 then 0.0 else x;
tel
```

See Contract Semantics for the formal reading of assumptions, guarantees, and modes.

34.19 Compositional and Modular Analysis

Contracts pay off in how a model is analyzed. By default Kind 2 analyzes a node against the full implementations of everything it calls, however deep the hierarchy runs. Two flags change that.

Compositional analysis, `--compositional true`, abstracts each call away by the callee's contract, so the analysis sees only what the contract promises instead of the callee's state. This can help Kind 2 reason about the system because the specification is likely simpler than the implementation.

However, using compositional mode, proving `top` correct with its callees abstracted says nothing about whether those callees honor their contracts. That is what **modular** analysis, `--modular true`, supplies: it analyzes every node in the hierarchy, bottom-up, and keeps going even when some node's properties are falsified.

The two are often used together:

```
kind2 --modular true --compositional true <file>.lus
```

Analyzed this way, each node is verified once against its own contract, and every caller uses that contract in place of the body. If a compositional analysis of `top` fails, the counterexample may be spurious — an artifact of the abstraction rather than a real defect — and when the callee has already been proved correct, Kind 2 *refines* the call, replacing the contract with the real implementation and analyzing again. A per-analysis limit is available as `--timeout_analysis`, alongside the global `--timeout`.

Two modifiers override these choices for an individual component. Writing `transparent` before `node` or `function` keeps it from being abstracted by its contract; writing `opaque` keeps it from being refined:

```
transparent function F(x: int) returns (y: int);
let
  y = x;
tel
```

See Techniques for how the two modes interact with each verification engine.

34.20 Realizability Checking

A contract can be wrong in a way no amount of implementation effort will fix: it can demand something impossible. The contract below cannot be satisfied by any implementation, because a negative `x` leaves no legal value for `y`:

```
node imported M(x: int) returns (y: int);
con
  guarantee 0 <= y and y <= x;
noc
```

Such a contract is **unrealizable**. **Realizability** asks whether a component can be built that, *for every* input sequence permitted by the assumptions, produces *some* output satisfying the guarantees.

Realizability checks are especially important for imported nodes and functions, as there is no implementation to witness realizability.

Kind 2 performs the check when the `CONTRACTCK` engine is enabled:

```
kind2 --enable CONTRACTCK <file>.lus
```

Kind 2 also checks the realizability of a node's *environment*—that the assumptions themselves can be met. Assumptions that no input sequence can satisfy make a node's guarantees vacuous.

When a contract is found unrealizable, Kind 2 shows a trace ending in a state from which the contract cannot be satisfied, together with the conflicting constraints (disable with `--print_deadlock false`).

See Contract Check for the remaining options.

34.21 More Examples

For more examples, see the Kind 2 web application at https://kind.cs.uiowa.edu/app/.

For the full language reference, see Kind 2 Input; the pages alongside it cover each type in depth, and Advanced Features covers what Kind 2 can do beyond proving properties invariant.

35 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:

    1. You must give any other recipients of the Work or Derivative Works a copy of this License; and
    2. You must cause any modified files to carry prominent notices stating that You changed the files; and
    3. 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
    4. 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 {2015-2022} {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.