Static Analysis via Horn Encoding from synchronous Dataflow Programs

Laure Gonnord, Szabolcs-Marton Bagoly, Lionel Morel

Analyse statique de propriétés de programmes Lustre via un encodage vers des Clauses de Horn

Szabolcs-Marton BAGOLY, Laure GONNORD, Lionel MOREL

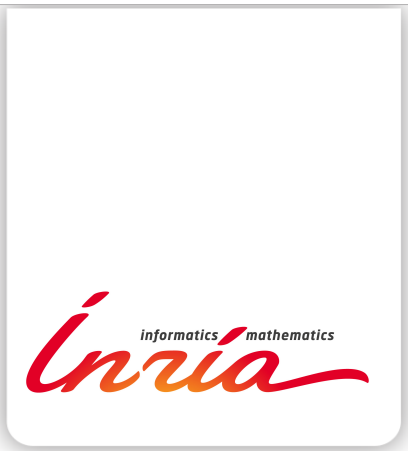
TECHNICAL REPORT

N° 0492

July 2017

Project-Teams SOCRATE and ROMA

ISSN 0249-0803 ISRN INRIA/RT--0492--FR+ENG



Analyse statique de propriétés de programmes Lustre via un encodage vers des Clauses de Horn

Szabolcs-Marton BAGOLY*, Laure GONNORD†, Lionel MOREL ‡

Project-Teams SOCRATE and ROMA

Technical Report n° 0492 — July 2017 — 25 pages

Abstract: In this technical report we define an encoding from Lustre programs to Array Horn Clauses.

Key-words: Static Analysis, Lustre, Horn Clauses, Array properties

* University of Cluj, Romania
† University of Lyon, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), F-69000 Lyon, France
‡ INSA de Lyon

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Static Analysis via Horn Encoding from synchronous Dataflow Programs

Résumé : Dans ce rapport nous étudions l'analyse statique de programmes Lustre via un encodage vers des clauses de Horn à Tableaux.

Mots-clés : Analyse Statique, Lustre, Clauses de Horn, Tableaux

Contents

1 Introduction

In this work, we present a method for verifying safety properties on Lustre [5] programs. The method consists of a translation of the Lustre programs into a popular representation used in program verification: Horn clauses with constraints. Different prover tools (Z3Prover [6], Vaphor [7]) are used to reason on the generated formula. Considering the improvements made on prover tools, we expect that they will successfully argument about the majority of the formulas generated from reduced complexity Lustre programs.

In Section 2 a brief introduction on the Lustre programming language and Horn clauses is given. Section 3 describes the contribution of our work, presenting the definitions of translating Lustre code into Horn clauses. Section 4 begins by briefly presenting the implementation details, continued by a discussion on the obtained experimental results. Section 5 concludes the report with the current limitations and future improvements that could be made.

2 Background

2.1 Basics of Lustre

Lustre [5] is a synchronous data flow programming language introduced by researchers at *Verimag* [1] in 1984. It is dedicated to the programming of reactive systems, particularly used in critical applications (power plants, airplanes, etc.). It is a data flow (every variable or expression denotes a sequence of values) and synchronous (programs are ruled by a clock) declarative programming language[8].

As presented in [3], in Lustre every variable or expression denotes a **flow**, i.e., a possibly infinite sequence of values of a given type. Conceptually, we can think about a Lustre program as an infinite loop, where each variable and expression takes their nth value of its sequence at the nth step in the loop. All usual arithmetic (`+,-,*,/,div,mod`), boolean (`or,and,not`), relational ($=, <>, <, <=, >, >=$) and conditional (`if..then..else..`) operators are provided.

There are two additional **temporal operators**, which provide the essential descriptive power of the language:

- the `pre` operator is used to talk about a value of a flow from the *previous step*. So, if X represents the sequence $X = (x_0, x_1, .., x_i, x_{i+1}, ..)$, then $pre(X) = (nil, x_0, x_1, .., x_{i-1}, x_i, ..)$.
- the $\rightarrow$ (followed by) operator was introduced to deal with the *nil* value from the head of the $pre(X)$ sequence. So, $Y = 0 \rightarrow pre(X)$ denotes the sequence of $(0, x_0, x_1, .., x_i, ..)$.

We define variables using *equations*: on the left having the variable which is defined by the expression on the right. Using `pre` operator recursive definitions of a variable may be provided.

Lustre programs are structured into **nodes**. A node is a system of equations having inputs, outputs and local variables (inside a node, local flows may be defined). Each output and local flow must be defined by exactly one equation, in any order. An equation may contain node calls, connecting nodes with each other. The Lustre program becomes a network of nodes, having a **main node** - the node of the program, which communicates with the "outside world": accepts inputs and produces outputs.

Figure 1 presents the graphical interpretation (x axis denotes the time) of the flows appearing in a node with two outputs: *sevseg* of type *int*, counting from 0 to 9. Whenever the input *reset* is set, the counter is set to it's initial value. The other output, *led_on*, is of type *bool* and whenever the input is not set negates it's previous state. The correspondent Lustre program is presented below.

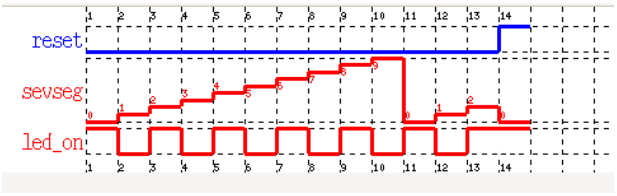


Figure 1: Lustre variables flow for the *cpt* node

```
node cpt(reset:bool) returns
(sevseg:  int; led_on: bool) ;
let
  sevseg = 0
 -> if (reset or pre(sevseg = 9))
                    then
 0 else pre(sevseg)+1;
  led_on = true -> if reset then
pre(led_on) else not(pre(led_on))
;
tel
```

Now, we present the syntax of a subset of Lustre used in this work, eliminating some of the advanced options of writing a Lustre program. We consider that a node has the following skeleton:

```
node node_name(in1_name:type;in2_name:type;..)
returns (out1_name:type;out2_name:type;..);
  [local variable declarations]
let
  {equations}
tel
```

A Lustre variable can be defined having the following type, user-defined data types being unsupported for the moment:

$\langle type \rangle ::= \langle simple\_type \rangle \mid \langle array\_type \rangle$

$\langle simple\_type \rangle ::=$ bool | int | real

$\langle array\_type \rangle ::= \langle simple\_type \rangle \text{ ^ } \langle size \rangle$

$\langle size \rangle ::=$ int

Note that we are dealing only with unidimensional arrays. Multidimensional arrays are beyond the scope of this work.

In section `equations` the following types of variable definitions are considered to exist:

$\langle x \rangle ::= \langle expr \rangle$

$\langle expr \rangle ::= \langle numexpr \rangle \mid var_1 \rightarrow var_2 \mid pre(var) \mid ite(bool\_var, var, var) \mid \langle iterator \rangle \mid$ nodecall

$\langle numexp \rangle ::=$ var | op var | var op var

$\langle iterator \rangle ::=$ map | red | fill | fillred | boolred

In the above definition, *var* can be one of the previously defined datatypes and `op` is one supported operator of the associated data type. For further information about the supported operations, the Lustre V6 Reference Manual[4] can be consulted. Of course, in a real program there are complex equations including several previously presented definitions, but we can always perform a split and transform them to fit the presented grammar.

Array iterators One of the main novelties in Lustre-V6 is to provide a notion of higher-order programming by defining **array iterators** to operate over arrays. In the following, the 5 five array iterators are presented (as in Lustre-V6 Reference Manual[4]). In Figure 2, a graphical interpretation of the iterators is presented.

- The **map** iterator transforms a scalar-to-scalar node/operator into an array-to-array node/-operator.

  Example 1. $map << +; 3 >> ([1,0,2],[3,6,-1]) \rightsquigarrow [4,6,1]$

- The **red** iterator transforms a scalar-to-scalar node/operator into an array-to-scalar node/-operator. The node argument must have a single output, a first input of the same type, and at least another input.

  Example 2. $red << +; 3 >> (0,[1,2,3]) \rightsquigarrow 6$

- The **fill** iterator transforms a scalar-to-scalar node/operator into a scalar-to-array node/-operator. The node argument must have a single input (input accumulator), a first output of the same type (output accumulator), and at least one another output.

  Example 3. $fill << incr; 4 >> (0) \rightsquigarrow (4,[0,1,2,3])$ *with*

```
node incr(ain : int) returns (aout, z : int);
let
    z = ain; aout = ain + 1;
tel
```

- The **fillred** iterator generalizes the fill and the red ones. It maps a scalar-to-scalar node into a "scalar and array" to "scalar and array" node. The node argument must have a (first) input and a (first) output of the same type, and at least one more input and one more output. The degenerated case with no other input (resp. output) corresponds to the fill (resp. red) iterators.

  Example 4. *A classical example is the binary adder, obtained by mapping the "full-adder". The unsigned sum Z of two bytes X and Y, and the corresponding overflow flag can be obtained by:* $(over, Z) = fillred << fulladd, 8 >> (false, X, Y)$*, where:*

```
node fulladd(cin, x, y : bool) returns (cout, z : bool);
let
  z = cin xor x xor y;
  cout = if cin then x or y else x and y;
tel
```

- The **boolred** iterator has 3 integer static input arguments: $boolred << i; j; k >>$ such that $0 \leq i \leq j \leq k$ and $k > 0$. It denotes a combinational node whose profile is $bool^k \rightarrow bool$, and whose semantics is given by: the output is true if and only if at least i and at most j elements are true in the input array.

  Example 5. *Two basic boolean array operations:*

  $\sharp(a_1, .., a_n) \rightsquigarrow boolred << 0, 1, n >> (a_1, .., a_n)$

  $nor(a_1, .., a_n) \rightsquigarrow boolred << 0, 0, n >> (a_1, .., a_n)$

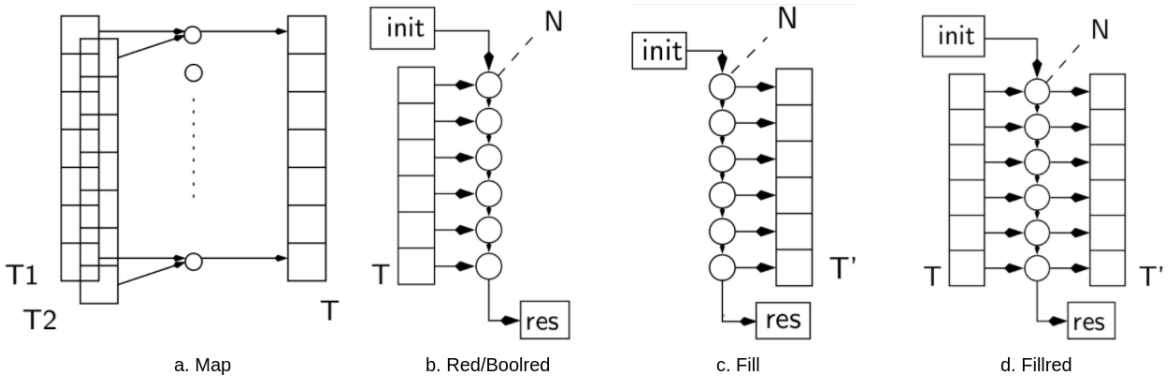


Figure 2: Array iterators in Lustre-V6

Observers In order to express safety properties in Lustre programs we adopt the technique of observers described in [3]. The desired properties are expressed through the use of observer nodes written in the same language. Basically, the observer node becomes the main node of the program, with the same inputs as the program, which is verified. The outputs of the verified program are stored in local flows of the observer node. The main node has a single boolean output representing the truth value of the property to verify. This output is computed by applying the property on the defined local flows.

Example 6. *Let us consider as an example the Lustre node graphically presented in Figure 1. Suppose, we want to add an observer, to monitor the value of sevseg variable, which should always remain between 0 and 9. To do this, the following observer node is added in the same program where the node cpt is defined.*

```
node obs(o_reset:bool) returns(ok:bool);
var
  o_sevseg:int;
  o_led_on:bool;
let
  o_sevseg,o_led_on = cpt(o_reset);
  ok = o_sevseg >= 0 and o_sevseg <= 9;
tel
```

2.2 Horn clauses

The target of our translation is to convert the initial Lustre program into Horn clauses, a widely used format for program verification. We aim to convert the program into Horn clauses by maintaining the majority of the types and operations from the source language.

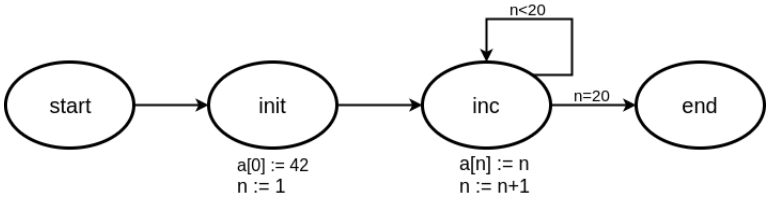


Figure 3: State automaton

The best way to explain the use of Horn clauses in this current work is by making an example: for now, let us not to use the semantics of Lustre and just consider a simple finite state automaton as presented in Figure 3. From state *start* without any modification, the system evolves to state *init*, where the first element of an array a is set to 42 and an index, n, is initialized. From state *init* the system evolves into a loop-like state, *inc*, where the rest of the array elements are initialized.

For each state from the automaton we consider a *state predicate* in Horn encoding with all variables used in the system, i.e., a,n. The format of the state predicate becomes: $state_x(a, n)$. For storing values into variables we use the following syntax: $state_{init}(a, n) \wedge a[0] = 42 \wedge n = 1$ meaning that we are in state *init*, where the first element of the array is set to 42 and n is set to value 1.

The previous systems' Horn clause description is, as follows:

$$\forall a, \forall n : state_{start}(a, n) \implies state_{init}(a, n)$$
$$\forall a, \forall n : state_{init}(a, n) \implies state_{inc}(a, n) \wedge a[0] = 42 \wedge n = 1$$
$$\forall a, \forall n : state_{inc}(a, n) \wedge n < 20 \implies state_{inc}(a, n') \wedge a[n] = n \wedge n' = n + 1$$
$$\forall a, \forall n : state_{inc}(a, n) \wedge n = 20 \implies state_{end}(a, n)$$

Considering the above formula, the verification $state_{end}(a, n) \wedge a[19] = 19$ should be *true*. There is only one element left to explain: how is the starting state, $state_{start}$ reached? The answer consists of adding the following clause to the formula:

$$\forall a, \forall n : true \implies state_{start}(a, n).$$

With this clause added, we know from the beginning that $state_{start}(..)$ is *true*, i.e., the initial state of the system is defined. From now on *implies* statement is written, as precondition should have or *true* or $\forall a, \forall n : state_{start}(a, n)$ in order to be reachable.

3 Contribution

In this section the effective translation of a Lustre program into Horn clauses is described, giving step by step definitions for each used Lustre syntactic element. We start by formalizing the translation of variables, expressions, programs with multiple nodes, observer nodes, and array iterators.

3.1 Translation of basic Lustre nodes into Horn clauses

Variables We define for each output and local variable a Horn state predicate. Currently, we do not define state predicates for the input variables as they are considered to exist and the valid before the execution of the program. Having a Lustre node with 2 output variables(x,y), in Horn encoding 2 different state predicates will be generated: $state_x$ and $state_y$. As parameters of a predicate, all the variables of the node will be included. The arity of a state predicate is given by

$$arity = N_{input\_vars} + N_{output\_vars} + N_{local\_vars} + 1, \tag{1}$$

i.e. the number of all variables + 1 (time-stamp counter - t, representing the tth step of execution).

We think about Lustre variables as an infinite sequence of values: when translating into Horn encoding, a simple Lustre variable becomes a unidimensional array of the same type. In Horn encoding, to show that we are dealing with an array, the following notation was adapted: a simple Lustre variable x, in Horn encoding becomes a_x. Similarly, a unidimensional array variable in Lustre will be translated as a 2-D array into Horn encoding.

Modeling the program flow In a Lustre program, the →(followed by) and *pre* operators are frequently used in order to use the value of a variable from previous step. The use of these operators rise the problem of variable initialization. In order to resolve the conflicts related to this problem, just as in case of state automaton, a **starting state** needs to be generated, with the time-stamp $t = 0$, meaning that the execution has not started yet, but when it starts, this is the place from where the first step should be generated:

$$\forall a_{in}, \forall a_{cond}, \forall a_x, \forall t : \ true \implies start(a_{in}, a_{cond}, a_x, 0)$$

In case of the →(followed by) operator, with the above predicate being true, the state predicate of the first step will be generated, containing the initial value of the output variables. It is important to mention, that the current output, will never be observable in the state in which it is set, only in the following one: so the first output, $a_x[0]$, is observable for the first time in:

$$state_x(a_{in}, a_{cond}, a_x, 1). \tag{2}$$

If the state predicate (2) has been generated in a clause, we say that $state_x$ is **valid** at $t = 1$.

Similarly, when we generate the translation of an arbitrary expression, we have to make sure that all the variables that appear in the right-side of the expression have their state predicates valid at moment $(t + 1)$, so we can access the array element at index t. This flow of variable

initialization is illustrated in Figure 4. Note that if the right-side of the expression contains only input variables, there is no need for variable validation. As they come from an external source, not being generated during the execution of the program, they are considered to be valid in every moment of the execution.

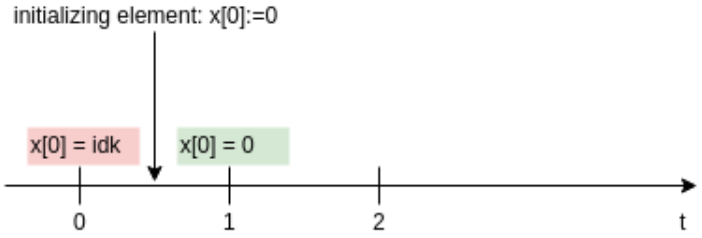


Figure 4: Variable value observability in Horn encoding of Lustre programs

Expressions Now, we propose an encoding of all variable definitions following a classical syntax directed translation. As an example, we consider a Lustre node with 2 input variables(*in, cond*) and one output variable: x, the total arity of a state predicate being 4. Respecting the previously mentioned consideration that in Horn encoding we associate one state predicate for each output and local variable, in the following definitions we will have one state predicate: $state_x$. Inputs are considered to be valid from the start of execution, so there is no need of making validations about them. Each equation below is, implicitely preceeded by $\forall a_{in}, \forall a_{cond}, \forall a_x, \forall t$:

- $x ::= var$ is translated with:

$$(t \geq 0) \implies state_x(a_{in}, a_{cond}, a_x, t+1) \wedge a_x[t] = var,$$

  where var may be an input, a literal or a temporary definition of a numerical expression. In this last case, we have to make sure that the value of var is valid in the corresponding moment. The previous translation becomes:

$$(t \geq 0) \wedge state_{var}(a_{var}, t+1) \implies state_x(a_{in}, a_{cond}, a_x, t+1) \wedge a_x[t] = a_{var}[t],$$

- $x ::= var_1\ op\ var_2$ is translated with:

$$(t \geq 0) \implies state_x(a_{in}, a_{cond}, a_x, t+1) \wedge a_x[t] = a_{var_1}[t]\ op\ a_{var_2}[t],$$

  where op is an operation, supported by the type of x, and var_1,var_2 are the operands (may be inputs, constant values, temporary definition of another numerical expressions). In the last case corresponding state validations should be made. If var_1 is missing we are dealing with an unary operation.

- $x ::= var_1 \rightarrow var_2$ is translated with:

$$\begin{gathered}(t = 0) \wedge start(a_{in}, a_{cond}, a_x, t) \wedge state_{var_1}(a_{var_1}, t+1) \implies \\ state_x(a_{in}, a_{cond}, a_x, t+1) \wedge a_x[t] = a_{var_1}[t] \\ (t > 0) \wedge state_{var_2}(a_{var_2}, t+1) \implies state_x(a_{in}, a_{cond}, a_x, t+1) \wedge a_x[t] = a_{var_2}[t]\end{gathered}$$

- $x ::= pre(var)$ is translated with:

$$(t >= 0) \wedge state_{var}(a_{var}, t) \implies state_x(a_{in}, a_{cond}, a_x, t+1) \wedge a_x[t] = a_{var}[t-1]$$

- $x ::= ite(bool\_var, var_1, var_2)$ has the following associated clause:

$$(t \geq 0) \wedge state_{bool\_var}(a_{in}, a_{bool\_var}, a_x, t+1) \wedge state_{var_1}(a_{var_1}, t+1) \wedge state_{var_2}(a_{var_2}, t+1) \implies state_x(a_{in}, a_{bool\_var}, a_x, t+1) \wedge a_x[t] = ite(a_{bool\_var}[t], a_{var_1}[t], a_{var_2}[t])$$

Example 7. *Using the above definitions, the translation of the following Lustre node is:*

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

$$\forall a_x, \forall t: \ true \implies start(a_x, 0)$$
$$\forall a_x, \forall t: \ t = 0 \wedge start(a_x, t) \implies state_x(a_x, t+1) \wedge a_x[t] = 0$$
$$\forall a_x, \forall t: \ t > 0 \wedge state_x(a_x, t) \implies state_x(a_x, t+1) \wedge a_x[t] = a_x[t-1] + 1$$

Connecting multiple nodes Usually, a Lustre program is constructed from multiple user defined interconnected nodes. From a main node, several secondary nodes may be called in order to obtain a partial result, then computing the output of the main node using the partial results from secondary nodes. One representative example, adapted after the example from [4], is the detection of falling edges in a stream of boolean inputs: to obtain a node, which is detecting falling edges, a node detecting rising edges may be defined; calling this node, with the negated inputs the falling edge detector node can be obtained.

Example 8. *Program to detect falling edges:*

```
node rising(x:bool) returns (y:bool);
let
  y = false -> x and not pre(x);
tel

node falling(x:bool) returns (y:bool);
let
  y = rising(not x);
tel
```

As one may observe, in order to make a call for an another node, we have to provide the correct inputs(parameters of the node call) and specify the variable(s) where the result(s) should be stored(left-side of the expression). We are not interested in accessing the local variables of the called node.

Having the above considerations we extend the our Horn clause translation model with a new type of predicate: let us call it *interface predicate*. Every node used in a Lustre program, should have an interface predicate, which arity may differ from the usual state predicate, because of leaving out the local declarations. An interface predicate consists of the nodes' inputs and it's outputs for every time-stamp, $t >= 0$. When should the interface predicate be validated? At the moment when all the outputs of the node are valid for the current step.

Let us consider the following Lustre node header with 2 inputs, 1 output and 1 local variable:

```
node do_work(m,n:int) returns (x:int);
var s:int;
```

The Horn clause which generates the interface of the node can be defined as follows:

$$t \geq 0 \wedge state_x(a_m, a_n, a_x, a_s, t+1) \implies interface_{do\_work}(a_m, a_n, a_x, t+1)$$

Whenever, in the calling node an expression appears which calls the second node, when generating the translation, we have to make sure that the called node's interface is valid at time-stamp $t+1$, with the correct inputs: so we can simply match the outputs of the called node, with the variables from the calling node.

Note that, if the parameters of the called node are computed in the calling node, we also have to make sure that their value is valid in the moment of calling the interface. The Horn clause generated from translating a node call expression ($z = do\_work(c, d)$) is:

$$t \geq 0 \wedge interface_{do\_work}(c, d, z, t+1) \implies state_z(c, d, z, .., t+1)$$

Note that in this case, we do not have to store anything, because the element is already defined in the called node, and the output of the called node is matched with the variable from the calling node.

Observer nodes Remember that in Section 2 we introduced observer nodes in order to express safety properties on programs. We execute the following steps to make program verifications:

- we consider a node, let us name it *obs*, as being the main node of the program. The node has a single boolean output, *ok*, representing the property which needs to be satisfied.
- from the *obs* main node a call is made to the node of the program which computes the result which follows to be verified.
- having the value of the result from the node call, the *ok* is computed applying the wanted property on the result.
- the last step is to generate the Horn clause, which implies that *ok* is true in every moment:

$$\forall t, \forall a_{ok} : state_{ok}(a_{ok}, t+1) \implies a_{ok}[t]$$

Example 9. *Consider the Lustre node from Example 7. If we wish to present an automatic translation of the node, first of all we have to transform the Lustre program into and intermediate representation, splitting the non-trivial expressions and then apply the transformation rules on the new program. The previous program becomes:*

```
node inc() returns (x:int);
var
  var_1:int;
  var_2:int;
let
  var_1 = pre(x);
  var_2 = var_1 + 1;
  x = 0 -> var_2;
tel
```

The Horn clause translation of this new node is (the universal quantifier declarations from now on will be omitted):

$$true \implies start(a_x, a_{var_1}, a_{var_2}, 0)$$

$$t \geq 0 \wedge state_x(a_x, a_{var_1}, a_{var_2}, t) \implies state_{var_1}(a_{num}, a_x, a_{var_1}, a_{var_2}, t+1) \wedge a_{var_1}[t] = a_x[t-1]$$

$$t \geq 0 \wedge state_{var_1}(a_x, a_{var_1}, a_{var_2}, t+1) \implies state_{var_2}(a_x, a_{var_1}, a_{var_2}, t+1) \wedge a_{var_2}[t] = a_{var_1} + 1$$

$$t = 0 \wedge start(a_x, a_{var_1}, a_{var_2}, t) \implies state_x(a_x, a_{var_1}, a_{var_2}, t+1) \wedge a_x[t] = 0$$
$$t > 0 \wedge state_{var_2}(a_x, a_{var_1}, a_{var_2}, t+1) \implies state_x(a_x, a_{var_1}, a_{var_2}, t+1) \wedge a_x[t] = a_{var_2}[t]$$

The interface of the node:

$$t \geq 0 \wedge state_x(a_x, a_{var_1}, a_{var_2}, t+1) \implies interface_{inc}(a_x, t+1)$$

We want to verify a safety property on this program: the output of the node always stays positive, i.e. $x >= 0$. In the same file we declare an observer node. From there a call is made to the node inc storing it's result into an auxiliary variable aux. Then we compute the output of the observer node, with the equation: $ok = aux >= 0$, i.e. the property we want to verify.

The Lustre description of the observer node:

```
node obs() returns(ok:bool);
  var aux:int;
let
  aux = inc();
  ok = aux >= 0;
tel
```

The observer node is translated into Horn encoding

$$t \geq 0 \wedge interface_{inc}(a_{aux}, t+1) \implies state_{aux}(a_{ok}, a_{aux}, t+1)$$
$$t \geq 0 \wedge state_{aux}(a_{ok}, a_{aux}, t+1) \implies state_{ok}(a_{ok}, a_{aux}, t+1) \wedge a_{ok}[t] = a_{aux} \geq 0$$

The following line is added to the end of the formula:

$$t \geq 0 \wedge state_{ok}(a_{ok}, a_{aux}, t+1) \implies a_{ok}[t]$$

Example 10. *Here is presented an example with generating the modulus of a number. If the number is positive then nothing happens, otherwise the negation of the number is returned. This way the output of the node "modulus" should always be positive. An observer node is attached to the program to verify this property.*

The Lustre program is:

```
node modulus(x:int) returns(xpos:int);
let
  xpos = if x >= 0 then x else -x;
tel

node obs(in:int) returns(ok:bool);
var aux:int;
let
  aux = modulus(in);
  ok = aux >= 0;
tel
```

The description of the same program in an intermediate language from which point an automatic translation is done:

```
1  node modulus(x:int) returns (xpos:int);
   var
3    var1:bool;
     var2:int;
5  let
     var1 = x >= 0;
7    var2 = -(x);
     xpos = if var1 then x else var2;
9  tel

11 node obs(in:int) returns (ok:bool);
   var
13   aux:int;
   let
15   aux = modulus(in);
     ok = aux >= 0;
17 tel
```

Let us now define the Horn encoding of the node "modulus":

$$t \geq 0 \implies state_{var1}(a_x, a_{xpos}, a_{var1}, a_{var2}, t+1) \wedge a_{var1}[t] = a_x[t] \geq 0$$

$$t \geq 0 \implies state_{var2}(a_x, a_{xpos}, a_{var1}, a_{var2}, t+1) \wedge a_{var2}[t] = -a_x[t]$$

$$t \geq 0 \wedge state_{var1}(a_x, a_{xpos}, a_{var1}, a_{var2}, t+1) \wedge state_{var2}(a_x, a_{xpos}, a_{var1}, a_{var2}, t+1) \implies state_{xpos}(a_x, a_{xpos}, a_{var1}, a_{var2}, t+1) \wedge a_{xpos}[t] = ite(a_{var1}[t], a_x[t], a_{var2}[t])$$

$$t \geq 0 \wedge state_{xpos}(a_x, a_{xpos}, a_{var1}, a_{var2}, t+1) \implies interface_{modulus}(a_x, a_{xpos}, t+1)$$

The translation of the observer node:

$$t \geq 0 \wedge interface_{modulus}(a_{in}, a_{aux}, t+1) \implies state_{aux}(a_{in}, a_{ok}, a_{aux}, t+1)$$

$$t \geq 0 \wedge state_{aux}(a_{in}, a_{ok}, a_{aux}, t+1) \implies state_{ok}(a_{in}, a_{ok}, a_{aux}, t+1) \wedge a_{ok}[t] = a_{aux}[t] \geq 0$$

To make the property verification one last statement, which validates the output of the observer node, should be added:

$$t \geq 0 \wedge state_{ok}(a_{in}, a_{ok}, a_{aux}, t+1) \implies a_{ok}[t]$$

3.2 Translation of Lustre nodes with arrays into Horn clauses

Representing arrays In this section we will define a representation in Horn encoding of array types defined in Lustre. As we already defined an array for a variable to represent simple variables, for a Lustre unidimensional array, we have a multidimensional(2-D) array in Horn encoding, from now on noted with $mda_{varname}$. So, if we want to access the element with $index = 1$ of the Lustre array a, at time-stamp t, we have to access the following element: $mda_a[1][t]$. In order to be able to iterate over the array, we should also store information about the size of the array and the current index. As the value of $size$ is fixed and known at compile time, there is no need to make a predicate parameter of it, we can simply insert the hard-coded value of size in the translation.

The value of *index*, may change at each step, so it will be defined as a predicate parameter. With this additional parameter in the state predicates, the arity of the states is different from the arity calculated using equation (1). The equation now becomes:

$$arity = N_{input\_vars} + N_{output\_vars} + N_{local\_vars} + N_{index\_vars} + 1, \tag{3}$$

where N_{index_vars} is equal to the number of different sized arrays.

Example 11. *Given the following header of a Lustre node,*

```
node array_node(start:bool, count:int)
returns(x:int^4);
```

the corresponding Horn clause state predicate is: $state_x(a_{start}, a_{count}, mda_x, index_x, t)$*, where the parameter* $index_x \in \{0, 1, 2, 3\}$.

Variable observability In favor of uniformity, we consider that only one array element will be initialized per clause. With every operation, we start from $index = 0$, and iterate over the elements of the array, until *index* reaches the value of *size*. This means that all the elements of the array were initialized. A state predicate of an array variable has the additional condition of $index = size$ in order to be **valid**.

3.2.1 Translation without abstraction

Map iterator Knowing that the size of the arrays part of the Map iterator must be equal, a single *index* and *size* parameter can be used. The Horn clause translation of a node containing a generalized map iterator with 2 inputs and 1 output, where the operation efectuated on the array elements is given by *map_op* and s is a hard-coded value:

$$t \geq 0 \wedge state_a(mda_a, mda_b, mda_x, s, t+1) \wedge state_b(mda_a, mda_b, mda_x, s, t+1) \implies state_x(mda_a, mda_b, mda_x, 0, t+1)$$

$$t \geq 0 \wedge state_x(mda_a, mda_b, mda_x, index, t+1) \wedge index < s \implies state_x(mda_a, mda_b, mda_x, index+1, t+1) \wedge mda_x[index][t] = map\_op(mda_a[index][t], mda_b[index][t])$$

Example 12. *Let us consider the following Lustre V6 node, which maps the +(plus) operator on the elements of the 2 input arrays(a,b) and stores the result in the output array(x).*

```
node usemap(a,b:int^2) returns (x:int^2);
let
  x = map<<+,2>>(a,b);
tel
```

The Horn clause encoding of the node contains a state predicate $state_x$ *and a predicate in order to store the size of the array:*

$$t \geq 0 \implies state_x(mda_a, mda_b, mda_x, 0, t+1)$$

$$t \geq 0 \wedge state_x(mda_a, mda_b, mda_x, index, t+1) \wedge index < 2 \implies state_x(mda_a, mda_b, mda_x, index+1, t+1) \wedge mda_x[index][t] = mda_a[index][t] + mda_b[index][t]$$

Red iterator The *red*(reduce) iterator reduces an array to a scalar variable. Note that in order to obtain the final result we have to iterate over the array and save the partial results in an addition *accumulator* variable. This can be done without introducing a new element in the parameter list of the state predicate, if we store the partial results in the same variable where the final result will be stored.

$$t \geq 0 \wedge state_{init}(a_{init}, mda_{arr}, a_x, index, t+1) \wedge state_{arr}(a_{init}, mda_{arr}, a_x, index, t+1) \implies state_x(a_{init}, mda_{arr}, a_x, 0, t+1)$$

$$t \geq 0 \wedge state_x(a_{init}, mda_{arr}, a_x, index, t+1) \wedge index = 0 \implies state_x(a_{init}, mda_{arr}, a_x, index+1, t+1) \wedge a_x[t] = red\_op(a_{init}[t], mda_{arr}[0][t])$$

$$t \geq 0 \wedge state_x(a_{init}, mda_{arr}, a_x, index, t+1) \wedge 0 < index \wedge index < s \implies state_x(a_{init}, mda_{arr}, a_x, index+1, t+1) \wedge a_x[t] = red\_op(a_x[t], mda_{arr}[index][t])$$

Example 13. *We consider the following Lustre V6 node, with the red array iterator, having the input of an integer type array. The +(plus) operator is applied over the Red iterator. The sum of the array elements added to the value of initial input should be computed. If init = 0, then the result is the sum of the array elements.*

```
node usered(init:int;arr:int^4) returns (x:int);
let
  x = red<<+,4>>(init,arr);
tel
```

The translation of the above node:

$$t \geq 0 \implies state_x(a_{init}, mda_{arr}, a_x, 0, t+1)$$

$$t \geq 0 \wedge state_x(a_{init}, mda_{arr}, a_x, index, t+1) \wedge index = 0 \implies state_x(a_{init}, mda_{arr}, a_x, index+1, t+1) \wedge a_x[t] = a_{init}[t] + mda_{arr}[index][t]$$

$$t \geq 0 \wedge state_x(a_{init}, mda_{arr}, a_x, index, t+1) \wedge 0 < index \wedge index < 4 \implies state_x(a_{init}, mda_{arr}, a_x, index+1, t+1) \wedge a_x[t] = a_x[t] + mda_{arr}[index][t]$$

When the value of index reaches 4, the current iteration has been terminated and the value of x is correctly set.

Fill iterator Respecting the constraints of our approach, which says that for each output and local variable we generate a state predicate, in the case of Fill iterator we have to generate multiple state predicate (for each individual output of the operation). The computation of a following element from the array needs that the partial result to be valid for the corresponding index. That means that, in a state predicate we not only have to store the value of the state predicate's variable, but we have to update all the variables, which take part in constructing the next element of the array.

The $res, x = fill << fill\_op, s >> (init)$ Lustre expression is translated as follows:

- state predicates for output x:

$$t \geq 0 \wedge state_{init}(a_{init}, mda_x, a_{res}, index, t+1) \implies state_x(a_{init}, mda_x, a_{res}, 0, t+1)$$

$$t \geq 0 \wedge state_x(a_{init}, mda_x, a_{res}, index, t+1) \wedge index = 0 \implies state_x(a_{init}, mda_x, a_{res}, index+1, t+1) \wedge (a_{res}[t], mda_x[index][t]) = fill\_op(a_{init}[t])$$

$$t \geq 0 \wedge state_x(a_{init}, mda_x, a_{res}, index, t+1) \wedge 0 < index \wedge index < s \implies$$
$$state_x(a_{init}, mda_x, a_{res}, index+1, t+1) \wedge (a_{res}[t], mda_x[index][t]) = fill\_op(a_{res}[t])$$

- state predicates for output *res*:

$$t \geq 0 \wedge state_{init}(a_{init}, mda_x, a_{res}, index, t+1) \implies state_{res}(a_{init}, mda_x, a_{res}, 0, t+1)$$

$$t \geq 0 \wedge state_x(a_{init}, mda_x, a_{res}, index, t+1) \wedge index = 0 \implies$$
$$state_{res}(a_{init}, mda_x, a_{res}, index+1, t+1) \wedge (a_{res}[t], mda_x[index][t]) = fill\_op(a_{init}[t])$$

$$t \geq 0 \wedge state_x(a_{init}, mda_x, a_{res}, index, t+1) \wedge 0 < index \wedge index < s \implies$$
$$state_{res}(a_{init}, mda_x, a_{res}, index+1, t+1) \wedge (a_{res}[t], mda_x[index][t]) = fill\_op(a_{res}[t])$$

Example 14. *The following program combines all the Lustre array iterators presented up until now. It has as input an array of integers and the maximum element from the array is selected. In order to make sure that the maximum selection was done correctly, the following property is verified: all elements of the array are less of equal with the selected maximum.*

The program is made of three Lustre nodes: a main observer node, where the property is computed and two auxiliary nodes used as operations in the array iterators. Note that in the below program has already been introduced the intermediate auxiliary variables to simplify the expressions:

```
-- observer node
-- guarantees that max is greater than every element of the array
node obs(in:int^4) returns (ok:bool)
var
  max0,max,var1:int;
  max_list:int^4;
  ok_list:bool^4;
let
  var1 = in[0];
  max = red<<selectMax;4>>(var1, in);
  max0,max_list = fill<<same,4>>(max);
  ok_list = map<< >=, 4>>(max_list,in);
  ok = red<< and,4>>(true,ok_list);
tel

-- returns the same input in two outputs
node same(m:int) returns (m0,m1:int);
let
  m0 = m;
  m1 = m;
tel

-- returns the maximum of the two inputs
node selectMax(c,d:int) returns (x:int)
var
  var2:bool;
let
```

```
28    var2 = c > d;
      x = if var2 then c else d;
30  tel
```

Now, let us define each node in Horn encoding starting with the "selectMax" node:

$$t \geq 0 \implies state_{var2}(a_c, a_d, a_x, a_{var2}, t+1) \wedge a_{var2}[t] = a_c[t] > a_d[t]$$

$$t \geq 0 \wedge state_{var2}(a_c, a_d, a_x, a_{var2}, t+1) \implies state_x(a_c, a_d, a_x, a_{var2}, t+1) \wedge a_x[t] = ite(a_{var2}[t], a_c[t], a_d[t])$$

$$t \geq 0 \wedge state_x(a_c, a_d, a_x, a_{var2}, t+1) \implies interface_{selectMax}(a_c, a_d, a_x, t+1)$$

Node "same" is translated with:

$$t \geq 0 \implies state_{m0}(a_m, a_{m0}, a_{m1}, t+1) \wedge a_{m0}[t] = a_m[t]$$

$$t \geq 0 \implies state_{m1}(a_m, a_{m0}, a_{m1}, t+1) \wedge a_{m1}[t] = a_m[t]$$

$$t \geq 0 \wedge state_{m0}(a_m, a_{m0}, a_{m1}, t+1) \wedge state_{m1}(a_m, a_{m0}, a_{m1}, t+1) \implies interface_{same}(a_m, a_{m0}, a_{m1}, t+1)$$

The observer node has the below translation. In order to keep the formulas simple we introduce the following notation: $p ::= mda_{in}, a_{ok}, a_{var1}, a_{max}, a_{max0}, mda_{ok\_list}, mda_{max\_list}$

$$t \geq 0 \implies state_{var1}(p, index, t+1) \wedge a_{var1}[t] = mda_{in}[0][t]$$

$$t \geq 0 \wedge state_{var1}(p, index, t+1) \implies state_{max}(p, 0, t+1)$$
$$t \geq 0 \wedge state_{max}(p, index, t+1) \wedge interface_{selectMax}(a_{var1}, mda_{in}[index], a_x, t+1) \wedge index = 0 \implies state_{max}(p, index+1, t+1) \wedge a_{max}[t] = a_x[t]$$
$$t \geq 0 \wedge state_{max}(p, index, t+1) \wedge interface_{selectMax}(a_{max}, mda_{in}[index], a_x, t+1) \wedge 0 < index \wedge index < 4 \implies state_{max}(p, index+1, t+1) \wedge a_{max}[t] = a_x[t]$$

The expression $max0, max\_list = fill << same, 4 >> (max);$ *is translated with 6 Horn clauses:*

$$t \geq 0 \wedge state_{max}(p, 4, t+1) \implies state_{max0}(p, 0, t+1)$$
$$t \geq 0 \wedge state_{max0}(p, index, t+1) \wedge interface_{same}(a_{max}, a_{m0}, a_{m1}, t+1) \wedge index = 0 \implies state_{max0}(p, index+1, t+1) \wedge mda_{max\_list}[index][t] = a_{m1}[t] \wedge a_{max0}[t] = a_{m0}[t]$$
$$t \geq 0 \wedge state_{max0}(p, index, t+1) \wedge interface_{same}(a_{max0}, a_{m0}, a_{m1}, t+1) \wedge 0 < index \wedge index < 4 \implies state_{max0}(p, index+1, t+1) \wedge mda_{max\_list}[index][t] = a_{m1}[t] \wedge a_{max0}[t] = a_{m0}[t]$$

$$t \geq 0 \wedge state_{max}(p, 4, t+1) \implies state_{max\_list}(p, 0, t+1)$$
$$t \geq 0 \wedge state_{max\_list}(p, index, t+1) \wedge interface_{same}(a_{max}, a_{m0}, a_{m1}, t+1) \wedge index = 0 \implies state_{max\_list}(p, index+1, t+1) \wedge mda_{max\_list}[index][t] = a_{m1}[t] \wedge a_{max0}[t] = a_{m0}[t]$$
$$t \geq 0 \wedge state_{max\_list}(p, index, t+1) \wedge interface_{same}(a_{max0}, a_{m0}, a_{m1}, t+1) \wedge 0 < index \wedge index < 4 \implies state_{max\_list}(p, index+1, t+1) \wedge mda_{max\_list}[index][t] = a_{m1}[t] \wedge a_{max0}[t] = a_{m0}[t]$$

Translation of $ok\_list = map << \geq, 4 >> (max\_list, in)$*:*

$$t \geq 0 \wedge state_{max\_list}(p, 4, t+1) \implies state_{ok\_list}(p, 0, t+1)$$
$$t \geq 0 \wedge state_{ok\_list}(p, index, t+1) \wedge index < 4 \implies state_{ok\_list}(p, index+1, t+1) \wedge mda_{ok\_list}[index][t] = mda_{max\_list}[index][t] >= mda_{in}[index][t]$$

The output of the observer node is computed with the last red iterator:

$$t \geq 0 \wedge state_{ok\_list}(p, 4, t+1) \implies state_{ok}(p, 0, t+1)$$
$$t \geq 0 \wedge state_{ok}(p, index, t+1) \wedge index = 0 \implies state_{ok}(p, index+1, t+1) \wedge a_{ok}[t] = (true \wedge mda_{ok\_list}[index][t])$$
$$t \geq 0 \wedge state_{ok}(p, index, t+1) \wedge 0 < index \wedge index < 4 \implies state_{ok}(p, index+1, t+1) \wedge a_{ok}[t] = (a_{ok}[t] \wedge mda_{ok\_list}[index][t])$$

In order to verify that the described property is satisfied or not the following line is added to the translation:

$$t \geq 0 \wedge state_{ok}(p, 4, t+1) \implies a_{ok}[t]$$

Fillred iterator Similarly to the *Fill* iterator, we have to define states for each generated output by the iterator. The $res, x = fillred << fillred\_op, s >> (init, arr)$ expression is translated, as follows:

- state predicates for output x:

$$t \geq 0 \wedge state_{init}(a_{init}, mda_{arr}, a_{res}, mda_x, index, t+1) \wedge state_{arr}(a_{init}, mda_{arr}, a_{res}, mda_x, s, t+1) \implies state_x(a_{init}, mda_{arr}, a_{res}, mda_x, 0, t+1)$$

$$t \geq 0 \wedge state_x(a_{init}, mda_{arr}, a_{res}, mda_x, index, t+1) \wedge index = 0 \implies state_x(a_{init}, mda_{arr}, a_{res}, mda_x, index+1, t+1) \wedge (a_{res}[t], mda_x[0][t]) = fillred\_op(a_{init}[t], mda_{arr}[0][t])$$

$$t \geq 0 \wedge state_x(a_{init}, mda_{arr}, a_{res}, mda_x, index, t+1) \wedge 0 < index \wedge index < s \implies state_x(a_{init}, mda_{arr}, a_{res}, mda_x, index+1, t+1) \wedge (a_{res}[t], mda_x[index][t]) = fillred\_op(a_{res}[t], mda_{arr}[index][t])$$

- state predicates for output *res*:

$$t \geq 0 \wedge state_{init}(a_{init}, mda_{arr}, a_{res}, mda_x, index, t+1) \wedge state_{arr}(a_{init}, mda_{arr}, a_{res}, mda_x, s, t+1) \implies state_{res}(a_{init}, mda_{arr}, a_{res}, mda_x, 0, t+1)$$

$$t \geq 0 \wedge state_x(a_{init}, mda_{arr}, a_{res}, mda_x, index, t+1) \wedge index = 0 \implies state_{res}(a_{init}, mda_{arr}, a_{res}, mda_x, index+1, t+1) \wedge (a_{res}[t], mda_x[0][t]) = fillred\_op(a_{init}[t], mda_{arr}[0][t])$$

$$t \geq 0 \wedge state_x(a_{init}, mda_{arr}, a_{res}, mda_x, index, t+1) \wedge 0 < index \wedge index < s \implies state_{res}(a_{init}, mda_{arr}, a_{res}, mda_x, index+1, t+1) \wedge (a_{res}[t], mda_x[index][t]) = fillred\_op(a_{res}[t], mda_{arr}[index][t])$$

Boolred iterator In case of the *boolred* iterator, we have to deal with additional parameters: *min* and *max*. Note that: $min \leq max \leq size$. As the values of the integers *min* and *max* must be known at compile time, we can use hard-coded values as we done with the array size. The Horn clause translation of the *boolred* iterator, having the $state_x(mda_{in}, a_x, acc, index, t)$ state predicate for an expression like $x = boolred << min; max; s >> (in\_bool)$ is:

$$t \geq 0 \wedge state_{in\_bool}(mda_{in\_bool}, a_x, acc, s, t+1) \implies state_x(mda_{in\_bool}, a_x, 0, 0, t+1)$$

$$t \geq 0 \wedge state_x(mda_{in\_bool}, a_x, acc, index, t+1) \wedge index < s \implies$$
$$state_x(mda_{in\_bool}, a_x, acc_{new}, index+1, t) \wedge acc_{new} = ite(mda_{in\_bool}[index][t], acc+1, acc)$$

$$t \geq 0 \wedge state_x(mda_{in\_bool}, a_x, acc, index, t+1) \wedge index = s \implies$$
$$state_x(mda_{in\_bool}, a_x, -1, index, t+1) \wedge a_x[t] = ite(min \leq acc \wedge acc \leq max, true, false)$$

Note that, in this case the arity of the state predicate does not respect the equation 3, because 1 additional parameter was included to store the intermediate results of total number of *true* values.

In order to be able to choose the state predicate, where the output is correctly initialized we make the following choice: when *index* reaches the value of *size*, we know that all elements has been processed and the accumulator is correctly set, so we can store the value of the output. To make difference between this and the state where the output is also initialized, we set the value of the accumulator to -1, which state normally should be unreachable.

In summary, in order to use the value of the *boolred* iterator's output we have to make sure that the state predicate of the output is valid with $index = size$ and $acc = -1$.

3.2.2 Translation with abstaction

Red iterator with abstraction As we observed, formulas generated from programs using red iterators have an exponential behaviour with respect to the size of array. Using the Vaphor to get rid of arrays we could diminiuate the execution time and memory usage.

After having a basic translation, in of hope making an *array size indepentent* translations on programs containing array iterators, experiments were made to invent some form of abstractization, particularly on the Red iterator. In this case, we could generate formulas from programs using the red iterator without taking into account the size of the array. This would lead to constant execution time of the proving process.

The idea behind the abstraction is, that we can prove if a property is *true*, if the abstracted formula proves to be satisfiable. In other case, we can not conclude anything about the formula. We generate two partial boolean results from different states of the iteration. The final truth value of formula is given by the conjunction of the two partial results. They are computed in two steps, as follows (graphical illustration in Figure 5):

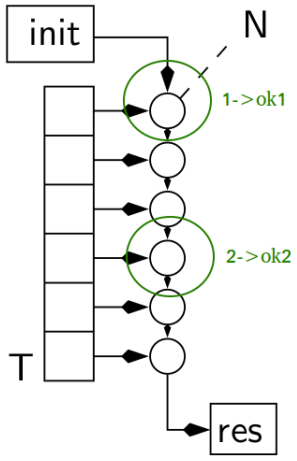


Figure 5: Abstraction of the red iterator

1. Verify the satisfiability of the property on the result computed by applying the operator on the *init* input variable and the *first element* of the array, considering that the used operands meet the *required preconditions*. From here the value of *ok1* is obtained.

2. We verify the property on a result computed in an arbitrary step,i, of the iteration by generating a result having as operands an *accumulator*, and the ith element of the array. We assume that the previous results are correct, i.e., the accumulator satisfies the desired property. The same must be true also about the current array element. From here the value of *ok2* is obtained. The final result $ok = ok1 + ok2$.

Example 15. *In this example the abstracted formula of a Red iterator is presented:*

```
..
r = red<<+;4>>(0,T);
ok = r >= 0;
..
```

*The ok property is the conjunction of two partial properties. In the below formula, elt*1 *and eltn are elements from the array T, which should satisfy the assumptions at the beginning of the program(if stated). They are stored in* $state_r$.

$$t \geq 0 \wedge state_{ok}(ok, t+1) \implies ok[t]$$

$$t \geq 0 \wedge state_{ok1}(r1, ok1, t+1) \wedge state_{ok2}(r2, ok2, t+1) \implies state_{ok}(ok, t+1) \wedge (ok[t] = ok1[t] \wedge ok2[t])$$

*Variables ok*1 *and ok*2 *are computed, if the corresponding results are valid:*

$$t \geq 0 \wedge state_{r1}(r1, t+1) \implies state_{ok1}(r1, ok1, t+1) \wedge ok1[t] = r1[t] \geq 0$$
$$t \geq 0 \wedge state_{r2}(r2, t+1) \implies state_{ok2}(r2, ok2, t+1) \wedge ok2[t] = r2[t] \geq 0$$

We define $r1 = 0 + elt1$*:*

$$t \geq 0 \wedge state_r(T, elt1, eltn, t+1) \implies state_{r1}(r1, ok1, t+1) \wedge r1[t] = 0 + elt1[t]$$

To compute $r2 = acc + eltn$ *we assume that acc already satisfies the property:*

$$t \geq 0 \wedge state_r(T, elt1, eltn, t+1) \wedge state_{ok2}(acc, ok2, t+1) \wedge ok2[t] \implies state_{r2}(r2, ok2, t+1) \wedge r2[t] = acc[t] + eltn[t]$$

As initially, no $state_{ok2}$ *is valid, we have to make the first step, and initialize the accumulator according to the property. Without this definition, because of the deadlock between the two above clauzes, the ok*2 *property will not be computed:*

$$t = 0 \wedge acc[t] \geq 0 \implies state_{ok2}(, t+1) \wedge ok2[t] = acc[t] \geq 0$$

4 Experiments

4.1 Setting

A prototype of the translation was implemented. The implementation does not contain any abstraction, we have a direct encoding of the Lustre semantics into Horn clauses. This way, the truth value of a tested property is directly binded to the satisfiability of the formula, i.e., *sat* means *property true* and *unsat* means that the property can not be true in the current context.

The OCaml code was inserted into the real compiler of Lustre-V6 developed by Verimag. An intermediate representation of the Lustre, called Lic(Lustre internal code), has been used as input. This representation is obtained from the original Lustre programs' source code after parsing, type checking, expression splitting, etc. It was important for us to have a program with the semantics very similar to the subset of Lustre presented in Section 2, maintaining the array expressions untouched, so the Lic program seemed to be an optimal input.

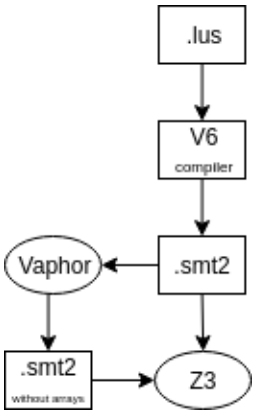


Figure 6: Setup

The prototype produces a SMTLIB2[2] format file. After that the Vaphor[7] and Z3Prover[6] tools are used to argument about the satisfiability of the generated formula. Experiments were made on a machine with 4 i7-4510U 2.00Ghz cores, 8 GiB RAM.

4.2 Results

Experimental results show that formulas generated from simple Lustre programs, without using arrays, can be almost instantly resolved by Z3. Note that, even if in the Lustre program we did not use arrays, after the translation all the variables become unidimensional arrays. However, for some programs, where complex safety properties are verified, the prover tends to give "unknown" or "timeout" ($>$ 5 min) results.

Bounded intervals Experiments show that the prover can easily reason about if a variable is bounded in a specified interval or not, if we try to reason using the extremities of the interval. However, for random variables, the execution of the prover times out.

Example 16. *For the below program, the result for different properties are presented:*

```
node obs(reset:bool) returns(ok:bool) ;
var
  sevseg:int;
let
  sevseg = 0 -> if (reset or pre(sevseg = 9))
                then 0 else pre(sevseg)+1;
  ok = sevseg < 10;
tel
```

- $ok = sevseg <= 9$; *gives* `sat`
- $ok = sevseg >= 9$; *gives* `unsat`
- $ok = sevseg <= 10$; *gives* `timeout`
- $ok = sevseg <= 0$; *gives* `unsat`
- $ok = sevseg <= 1$; *gives* `timeout`
- $ok = sevseg >= 1$; *gives* `unsat`

Prove sat vs. unsat During the experiments I have discovered, that sometimes it is easier to prove the unsatisfiability of a formula than it's satisfiability. So, in order to determine whether a formula F is satisfiable or not, we can reason about *not* F. If *not* F is an unsat formula, we conclude the correctness of F.

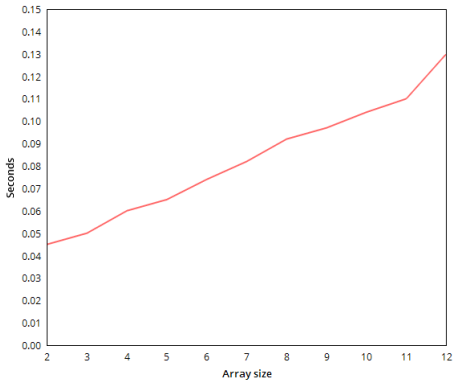


Figure 7: Simple node with Map iterator

Array iterators The majority of the experiments were done over programs using array iterators. For different programs, the generated formula a linear/exponential increase, with respect to the size of the used arrays, is noticeable. For formula generated from a simple program presented in example 17 we notice a linear increase, as presented in Figure 7.

Example 17.
```
node obs(a,b:int^100) returns (ok:bool);
  var x:int^100;
let
  x = map<<+,100>>(a,b);
  ok= x[100-1]+1 > a[100-1]+b[100-1];
tel
```

A different behaviour was observed in case of the red iterator. Example 18 uses the red iterator to compute the result which is verified. We can easily observe the exponential behaviour of the prover on the translated code, as presented in Figure 8. For *array index* = *16*, the test failed with **out of memory** error.

Example 18.
```
node add(a,b:int) returns (y:int);
var apos,bpos:int;
let
    apos = if a > 0 then a else 0;
    bpos = if b > 0 then b else 0;
    y = apos + bpos;
tel

node obs(init:int;c:int^10) returns(ok:bool);
var x:int;
let
    x = red<<add,10>>(init,c);
```

```
    ok = x >= init;
14 tel
```

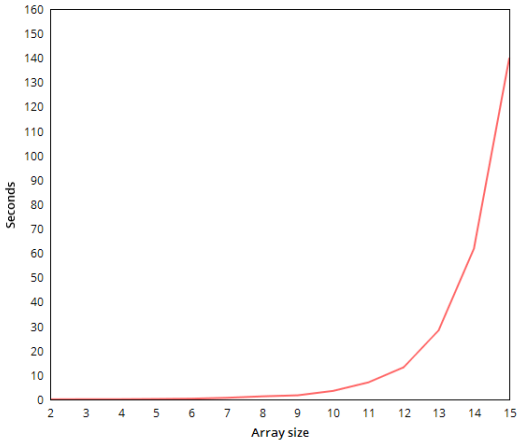


Figure 8: Red using Z3

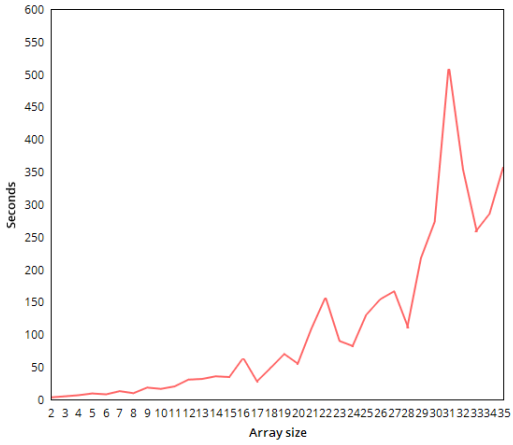


Figure 9: Red using Vaphor+Z3

In hope of reducing the complexity, the same program was fed to Vaphor[7] to abstract the arrays from the formula. We should notice a decrease in the prover's execution time, which is confirmed by the Figure 9. Note that using Vaphor, we were able to efectuate test to array sizes way above 16, the prover using *significantly less memory* on the abstracted formulas than before. A comparison between the execution time on the two formulas is given in Figure 10.

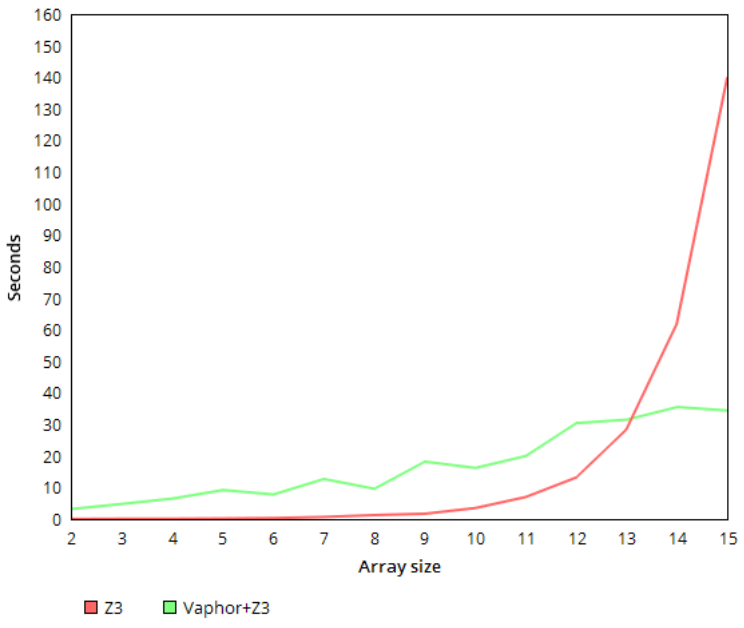


Figure 10: Comparison on execution time: Z3 vs. Vaphor+Z3

After observing the exponential behaviour on a formula containing the red iterator, it is

normal to expect the same behaviour from programs combining multiple iterators, such as the one presented in example 19, which determines the maximum value from an array of integers.

Example 19. *This Lustre program determinates the maximum from an array of integers. Then it is verified that the value of max is greater of equal than every element of the array.*

```
node obs(array : int^4) returns (ok:bool)
var maxl:int^4;max0:int;ok_list:bool^4;max:int;
let
  max = red<<selectMax;4>>(array[0], array);
  max0,maxl = fill<<same,4>>(max);
  ok_list = map<< >=, 4>>(maxl,array);
  ok = red<< and,4>>(true,ok_list);
tel

node same(m:int) returns (m0,m1:int);
let
  m0 = m;
  m1 = m;
tel

node selectMax(a,b:int) returns (x:int)
let
  x = if a > b then a else b;
tel
```

5 Conclusion

In this report, we have proposed a method to verify safety properties on programs written in Lustre-V6 (including array iterators) by translating the original program into a formula of Horn clauses, which is then sent to a prover. A prototype translator was implemented inside the Lustre-V6 compiler generating Horn clauses from the internal representation of Lustre programs (Lustre Internal Code). We have used Vaphor to generate abstracted formulas (without arrays) and the stock version of Microsoft Research Z3Prover to reason on the formulas. While the prover was able to solve many of the generated formulas within seconds and formulas from programs with array iterators within minutes (for larges array sizes), some of the formulas still could not be solved in reasonable time ("timeout" error).

As experiments show, the simple syntactic translation done is this work does not produce great results applied on programs with larger sized arrays. The proving time becomes too long and memory usage too big. Using the Vaphor to make abstraction of arrays in the formula, we were able to successfully reason about programs with array iterators using larger sized arrays ($>$15), because the prover uses way less memory when solving formulas without arrays.

Pushing forward the experiments on array iterators, an abstraction of the Red iterator was formalized, eliminating entirely the use of multidimensional arrays in the generated formula. Hand-written abstract translations of examples were tested. Including an automatic translation of the array iterators into abstracted (2-D array free) formulas would be a significant improvement on the presented work.

References

[1] Verimag Research Center. `http://www-verimag.imag.fr/`.

[2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[3] Albert Benveniste, Paul Caspi, Stefan A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

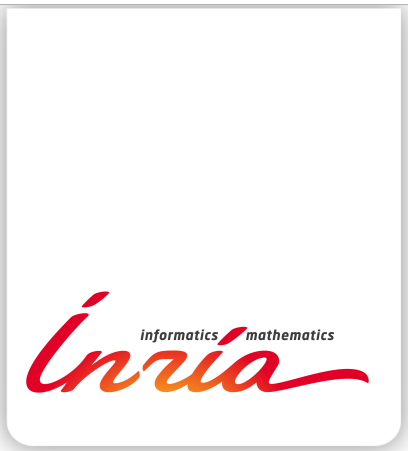
[4] Erwan Jahier, Pascal Raymond, Nicolas Halbwachs. The Lustre V6 Reference Manual. `http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf`.

[5] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[6] Microsoft Research. Z3 Theorem Prover. `https://github.com/Z3Prover/Z3`.

[7] David Monniaux and Laure Gonnord. Cell morphing: from array programs to array-free Horn clauses. In Xavier Rival, editor, *23rd Static Analysis Symposium (SAS 2016)*, Static Analysis Symposium, Edimbourg, United Kingdom, September 2016.

[8] Lionel Morel. Efficient compilation of array iterators for lustre. *Electronic Notes in Theoretical Computer Science*, 65(5):19 – 26, 2002. SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of ETAPS 2002).



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803