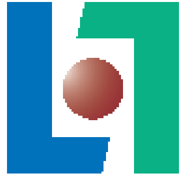
hal-00349898, version 1 - 5 Jan 2009



Laboratoire Bordelais de Recherche en Informatique
UMR 5800 - Université Bordeaux 1, 351, cours de la Libération,
33405 Talence CEDEX, France

Research Report `RR-1415-06`

On the partial translation of Lustre programs into the AltaRica language and vice versa

Alain Griffault, Gérald Point
`{griffaul,point}@labri.fr`

November 2, 2006

hal-00349898, version 1 - 5 Jan 2009

hal-00349898, version 1 - 5 Jan 2009

On the partial translation of Lustre programs into the AltaRica language and vice versa

Alain Griffault, Gérald Point
{griffaul,point}@labri.fr

November 2, 2006

Abstract

Lustre (1984) and AltaRica (2000) are two languages used to describe critical systems. The first one is a data-flow programming language; and its main purpose is the writing of formally verified programs implemented on embedded hardware. The second language has been designed mainly for the modelling and the analysis of non-specific systems. These languages are supported by different toolboxes and software workbenches which motivates the development of translators between the two languages.

In this report we describe a way to translate Lustre models into AltaRica ones (and vice versa). With some restrictions on the input model, the translation produces a model whose semantics is very closed to the original.

This work has been granted by the CERT-ONERA in the context of the ISAAC European Project.

Contents

1 Introduction **2**

2 The ALTARICA language **2**
- 2.1 Basic AltaRica . . . 3
- 2.2 Dialects and extensions . . . 5

3 The LUSTRE language **5**

4 The ARC tool **7**

5 ALTARICA to LUSTRE **9**
- 5.1 Restrictions on ALTARICA models . . . 9
- 5.2 Translation of leaf nodes . . . 10
- 5.3 Translation of hierarchical nodes . . . 15
- 5.4 Domains, signatures and global constants . . . 17
- 5.5 Properties of the translator . . . 19

6 LUSTRE to ALTARICA **20**
- 6.1 Restrictions on Lustre programs . . . 20
- 6.2 Types, functions and constants . . . 21
- 6.3 Variables, local variables and parameters . . . 22
- 6.4 Translation of subnode calls . . . 23
- 6.5 Translation of the `pre` operator . . . 24
- 6.6 Translation of the “followed by” operator (`->`) . . . 25

6.7 Translation of the `current`/`when` operators . . . 26
6.8 Modelling and synchronization of clocks . . . 26

7 Conclusion **27**

A Customization of the translations **27**
A.1 Customization of the Lustre to AltaRica Translation . . . 28
A.2 Customization of the AltaRica to Lustre Translation . . . 29

1 Introduction

Context The ISAAC (Improvement of Safety Activities on Aeronautical Complex systems : FP6-2002-Aero-1-501848) project extends the results of ESACS (Enhanced Safety Assessment for Complex Systems) that has shown the benefit of using formal techniques to assess aircraft safety. Its main goal is to go a step further into the improvement and integration of safety activities of aeronautical complex systems. Potential benefits range from higher confidence in the safety of systems to increased competitiveness of European industries. To reach the goals mentioned above, ISAAC will focus on the following three dimensions:

- Extension of formal techniques to deal with human error, common cause analysis, mission analysis, and testability. This will help providing a more comprehensive tool-supported coverage of the safety analysis process.
- Improvement of the ESACS notations to represent safety requirements and qualitative (timing) and quantitative aspects. This will help improving interaction and increasing the quality of the results provided by tools.
- Integration, achieved through common methodological recommendations and shared libraries and interfaces. This is one of the keys to improve, e.g, the efficiency of industrial processes, that more often rely on the use of different tools.

ISAAC results will be used by the partners to improve their safety process for their present and future programs. The results will also be disseminated to other areas, like, e.g., transportation (railway and automotive), industrial process control, energy production. The ISAAC consortium comprises aeronautical industries (Alenia, AIRBUS, Saab, SIA, Dassault) and research centres leaders in formal verification, safety assessment, and tool development (ITC, ONERA, OFFIS, PROVER).

The translators from ALTARICA to LUSTRE and vice versa have been designed and developped to allow :

- Collaboration between differents partners of the project.
- Comparaison between differents tools and techniques.

This work was done by the LaBRI as a sub-contractor of ONERA.

Acknowledgment Thanks to Remy Gobard who has done the first prototype of the ALTARICA to LUSTRE translator in Java during his master thesis.

2 The ALTARICA language

The ALTARICA language[4, 1] is a modelling language not dedicated to some specific domain and, moreover, it has been basically designed, in close partnership with industrials, to allow the description of both functional and dysfunctional aspects of systems. Here we just give a survey of the language; a complete description of ALTARICA can be found in [4] (its formal semantics is the main issue of [1]).

hal-00349898, version 1 - 5 Jan 2009

hal-00349898, version 1 - 5 Jan 2009

2.1 Basic AltaRica

An ALTARICA system is a collection of `nodes` organized hierarchically. While the root node of the hierarchy is the model of the whole system, the leaf nodes are its elementary components or functions. Each non-leave node controls its subnodes by means of interaction constraints (see below).

An AltaRica `node` is essentially a user-friendly description of a *constraint automaton*[5] which describe the evolution of two sets of variables (states and flows) on the occurence of events. The state variables, whose assignments represent the state of the node, evolve on the occurence of *stimuli* represented by the events. The flow variables represent an interface between the node and its environment, and can be considered as shared variables. The assignments of flow variables are expressed by means of constraints on both sides of the interface.

The following example shows an AltaRica node describing a switch. The three first lines are declarations of flows, states and events; then comes the description of behavourial elements:

Assertions (keyword `assert`) are Boolean conditions which must be satified by all state and flow assignment. These conditions express relations between the current state of the node and its flows. Here, the assertion of the switch expresses that when the switch is closed its flows must be equal (i.e. the switch does not block the traffic between `f1` and `f2`).

Transitions (keyword `trans`) express the evolution of state variables. Each transition has the form G `|- ` e `->` $\vec{s}$ `:=` $\vec{\alpha}$ where G is a Boolean condition over flows and states, e is the name of the event enabling the transition and $\vec{s}$ `:=` $\vec{\alpha}$ is a parallel assignment of state variables. A transition is enable if and only if the three following conditions are satisfied:

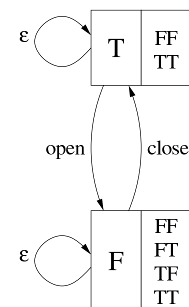
1. the event e occurs;
2. the *guard* G is satisfied by the current assignment of variables.
3. the assertions are not violated by the assignment $\vec{s}$ `:=` $\vec{\alpha}$.

If a transition is enabled then the next assignment of state variables can be realized.

On the right side of the node below is depicted its state graph. Each vertex of the graph consists in two boxes; the left one is the value of the `is_closed` state variable and the right hand side box is the authorized assignments for the flow variables. The reader should note the existence of two ε-labelled transitions. These latter are implicit and model two phenomena:

1. the *idle* action: the node can stay in its current state without doing any action;
2. the environment action: the node shares the control of flows with its environment and thus, these latter can change without any action of the node.

```
node Switch
  flow
    f1, f2 : bool;
  event
    open, close;
  state
    is_closed : bool;
  assert
    is_closed => (f1 = f2);
  trans
    not is_closed |- close -> is_closed := true;
    is_closed |- open -> is_closed := false;
edon
```



Each ALTARICA node may embed sub-nodes. This hierarchy allows the user to refine the description of nodes or to create models from the physical layout of the system.

Like the others, embedding nodes have their own behaviours described by means of state and flow variables, events, transitions and assertions. However their transitions and assertions may refer to sub-node interfaces and, subnode events may be synchronized. In a sense the embedding nodes apply some kind of control on their sub-nodes.

The following example shows a small power `System` composed with a `Generator`, the previous `Switch`, a `Consumer` (e.g a lamp) and a `Human`. The description of the embedding node `System` starts with the declaration of the 4 sub-nodes (`G`, `S`, `C` and `H`). The connectivity between the components is expressed with two assertions (see figure 1): the first one enforces the generator to supply power to the `f1` flow of the switch and the second assertion enforces the switch to propagate the power toward the consumer (remember that assertions are always satisfied by the system).

Finally we describe the action of the human on the switch. This is done by means of synchronization vectors which express the enforced simultaneity of its components: for example, the first vector expresses that when the operator (`H`) sets the power on, the switch must be closed.

```
node Switch ... edon

node Generator
  flow power : bool;
  assert power = true;
edon

node Consumer
  flow input_power : bool;
  ...
edon

node Human
  event set_power, unset_power;
  trans true |- set_power, unset_power -> ;
edon
```

```
node System
  sub G : Generator;
      S : Switch;
      C : Consumer;
      H : Human;
  assert
      G.power = S.f1;
      S.f2 = C.input_power;
  sync
      <H.power_on,S.close>;
      <H.power_off,S.open>;
edon
```

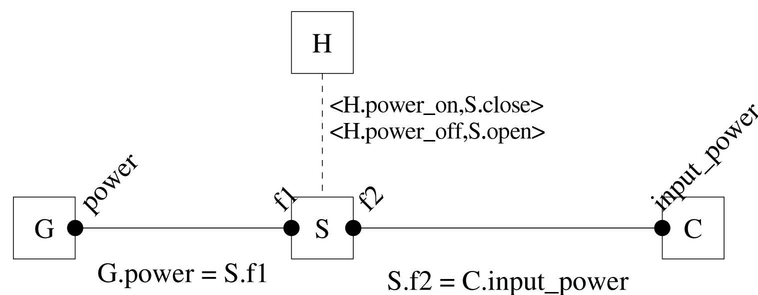


Figure 1: Connection between components are expressed by means of constraints.

Beyond this basic features, the language allows the use of:

Priorities: Each node can declare a partial order on its events. If, in some state, two transitions can be triggered then only those with a maximal event (w.r.t. the order) are actually triggered.

Weak synchronization vectors: Synchronization vectors express strong synchrony between their components i.e. if an event is not possible then the synchronization is not possible. The language permits to tag some components of the vectors with question mark (`?`). A tagged event is synchronized only if it is possible. Moreover, the user can indicate, for each vector, the number of tagged events that must be possible in order to enable the synchronization.

hal-00349898, version 1 - 5 Jan 2009

hal-00349898, version 1 - 5 Jan 2009

2.2 Dialects and extensions

All notions/concepts presented in the previous section were defined in [1, 4]. In order to deals with some misleadings or to improve the tools, some dialects have been designed:

AltaRica Finite Automata limits the values of the variables to finite domains. To be more and more user-friendly, attributes can be associated with variables and events, and data structures, parameters and arrays can be defined.

AltaRica Data Flow limits the values of the variables to finite domains, flow variables must be oriented (in, out) and no circular dependencies must occur.

Timed AltaRica introduces clocks in the AltaRica Finite Automata language.

AltaRica Abstract Data Types permits definition of external data types to extend AltaRica Finite Automata.

The first and last dialects have been merged to form the current version of used by the LaBRI. This version accepts the following features:

Visibility attributes: The genuine language enforces strict visibility constraints on events and variables. The user can bypassed default rules by adding a visibility attribute (`parent`, `public`, `private`) to the object (event or variable).

Compound types: Now on, the language permits the use of variable whose value is not necessarily a scalar but can be a structure or arrays (like in programming languages).

Parameters: These are constants that are not valued. They can be global or local to a node and they can be used in expressions or type declaration.

Abstract data types and function signatures: The language allows the declaration of `sorts` which are data types that must be interpreted by tools. Once they are declared, `sorts` can be used anywhere a type is required.

To manipulate abstract types, prototypes of functions can also be declared; for each function the user specify its name, the type of its arguments and the type of its return value.

3 The LUSTRE language

LUSTRE[2, 3] is a programming data-flow language defined at the IMAG Institute since 1984. LUSTRE compilers translate this high-level language into *target* languages (e.g. C ANSI or ADA) in order to be actually compiled into runnable programs or to be implemented on hardware. Here we present the main aspects of this language; a complete description can be found in [2].

In the LUSTRE language any expression is a *flow* which means a sequence of values indexed by a *clock* representing discrete instants. Flows can take their values into Booleans, integers, reals and abstract sets. These latter refer to types of the target language and their meaning is absolutely ignored by the LUSTRE semantics. The language allows classical operators on its basic types and permits the declaration of *external* functions (related to the target language) for abstract types. Basically any expression functionnally depends on the value of its variables:

Cycle	1	2	3	4	5	...
1	1	1	1	1	1	...
X	1	1	2	2	1	...
Y	2	0	2	2	0	...
X+Y+1	4	2	5	5	2	...

In order to create flows depending on program cycles, LUSTRE allows to build slower or faster clocks (never faster than the basic one) and to refer in expressions to the past values of flows:

hal-00349898, version 1 - 5 Jan 2009

- The "previous" operator (noted `pre`) memorizes the value of its argument at the previous cycle. If $e_1, e_2, \ldots$ is the sequence of values of a flow e then the flow pre(e) has the sequence of values: *nil*, $e_1, e_2, \ldots$ i.e. the sequence is shifted on the right. This operation introduces an *undefined* value called *nil*.

Cycle	1	2	3	4	5	6	...
X	1	1	2	2	1	4	...
pre(X)	*nil*	1	1	2	2	1	...
pre(pre(X))	*nil*	*nil*	1	1	2	2	...
pre(pre(X))+pre(X)	*nil*	*nil*	2	3	4	3	...

- The "followed by" operator (noted `->`) allows to define or to change the initial value of a flow. This operator is typically used to initialize an expression referring to the past cycle (by the use of the `pre` operator). If e_i and f_i ($i \geq 1$) are the values of flows e and f then the flow e -> f has the sequence of values: $e_1, f_2, f_3, \ldots$ (f is not shifted; f_1 is replaced by e_1).

- The `when` operator allows to sample a sequence with a Boolean flow I. The resulting sequence takes its values when the Boolean flow is `true`; furthermore, this resulting flow is based on a clock slower than the one of I.

- The `current` operator allows to "accelerate" a flow with a faster clock. A Boolean flow B defines a clock by its `true` values; B is itself based on some clock C. Now, given a flow X based on the clock B, current(X) is a flow based on C whose value is the value of X the last instant B was `true`.

Cycle	1	2	3	4	5	6	...
I	false	true	false	true	false	true	...
X	2	1	5	2	4	1	...
X when I		1		2		1	...
current(X when I)	*nil*	1	1	2	2	1	

A LUSTRE program is structured by means of *nodes*. A node represents an I/O machine taking as argument a set of flows identified by variables and returning another set of flows. Each output flow is defined by means of one equation implying input, local or other output flows (of course, an output flow can not depend, directly or not, on itself). The following node compute the number of times its input is true:

```
node counter(i : bool) returns (c : int);
let
   c = if i
         then 1 -> pre(c)+1
         else 0 -> pre(c);
tel.
```

A LUSTRE program can be described hierarchically using node calls; a node can use another node as a sub-function. The following example shows a node `Main` whose behaviour has been refined into 3 modes. An input flow indicates which `mode` is active and the output of `Main` is defined according to the selected mode.

```
node MODE_1(i : int) returns (o : int); let ... tel.
node MODE_2(i : int) returns (o : int); let ... tel.
node MODE_3(i : int) returns (o : int); let ... tel.

node Main(mode : int, i : int) returns (o :int);
let
   o = if mode = 1 then MODE_1(i)
       else if mode = 2 then MODE_2(i)
       else MODE_3(i);

   assert( 1 <= mode and mode <= 3 );
tel.
```

The reader should note the assertion on `mode` which ensures that the input of `Main` never goes out of the range $[1,3]$.

4 The ARC tool

ARC is the acronym for "**A**lta**R**ica **C**hecker". It is a command-line tool used to analyze AltaRica models. ALTARICA/LUSTRE translators are commands of this tool.

The ARC tool is freely available on the web site of the AltaRica project. It requires the installation of two additional libraries (`ccl` and `arsyntax`) also available on the site. Follow the download and installation instructions on the page: `http://altarica.labri.fr/Tools/ARC`.

Lustre-to-AltaRica translator: The ARC tool is able to read directly LUSTRE files; it translates the whole program (constants, function and nodes) into the ALTARICA language. Using the ARC command called `dump`, the user can see the translation of any loaded LUSTRE node.

Example 1 *The following example loads into* ARC *a hierarchical program with a toplevel node called* `Test`*. The tool checks if the filename terminates with the* `.lus` *extension; in this case, the translator is automatically applied to the* LUSTRE *program and all* LUSTRE *nodes are translated.*

The user can apply ARC *commands to any translated node. In this example, the* `flatten` *command is applied to the toplevel node* `Test` *(the command displays the equivalent* ALTARICA *node once the hierarchy has been removed) and also, the* `ts_marks` *command which computes a set of basic properties on the transition system of the node (this command is available on this model because the* `int` *type has been translated as a finite range; see section 6.2).*

```
point@laptop: cat rr-flat.lus
node NF1(i1,i2 : int) returns (o:bool);
let
   o = (i1 = 2*i2+3);
tel.

node NF2(i1,i2 : int) returns (o:bool);
let
   o = (i1 = 3*i2+2);
tel.

node Test(i1,i2 : int) returns (o : bool);
let
   o = #(NF1(i1,i2),NF2(i1,i2));
tel.

point@laptop: arc rr-flat.lus
Loading file 'rr-flat.lus'.
L2A: warning: setting integers to [0,10]
L2A: translating lustre node 'NF1'.
L2A: translating lustre node 'NF2'.
L2A: translating lustre node 'Test'.
arc>flatten Test
// flat semantics of node Test
node Test
  flow
   /* 1 */ 'sc_1.o' : bool : out;
   /* 2 */ 'sc_0.o' : bool : out;
   /* 3 */ o : bool : out;
   /* 4 */ 'sc_1.i1' : [0,10] : in;
   /* 5 */ 'sc_1.i2' : [0,10] : in;
```


hal-00349898, version 1 - 5 Jan 2009

```
    /* 6 */ 'sc_0.i1' : [0,10] : in;
    /* 7 */ 'sc_0.i2' : [0,10] : in;
    /* 8 */ i1 : [0,10] : in;
    /* 9 */ i2 : [0,10] : in;
   event
    /* 1 */ '<clock, sc_0.clock, sc_1.clock>';
    /* 2 */ '<$, sc_0.$, sc_1.$>';
   trans
    true |- '<clock, sc_0.clock, sc_1.clock>' -> ;
    true |- '<$, sc_0.$, sc_1.$>' -> ;
   assert
    'sc_1.o'=('sc_1.i1'=3*'sc_1.i2'+2);
    'sc_0.o'=('sc_0.i1'=2*'sc_0.i2'+3);
    'sc_0.i1'=i1;
    'sc_0.i2'=i2;
    'sc_1.i1'=i1;
    'sc_1.i2'=i2;
    o=(not 'sc_0.o' and 'sc_1.o' or not 'sc_1.o');
   // no initialization is specified.
edon
arc>ts_marks Test
/**
 * Cardinality of the properties computed for the transition system
 * of node 'Test'

 * State properties :
 * ------------------
 * any_s = 121
 * initial = 121
 *
 * Transition properties :
 * -----------------------
 * any_t = 29282
 * epsilon = 14641
 * self = 242
 * self_epsilon = 121
 * not_deterministic = 29282
 */
arc>
```

AltaRica-to-Lustre translator: The translation of ALTARICA models into LUSTRE is done by the `to_lustre` command as shown by the following ARC session.

Example 2 *This example loads into* ARC *a* `Main` *node built with 3 "empty" subnodes. The* `to_lustre` *command is used to display the translation of the* `Main` *node.*

```
point@laptop: cat rr-empty-subnode.alt
node EmptyNode
edon

node Main
  sub a1, a2, a3: EmptyNode
edon

point@laptop: arc rr-empty-subnode.alt
Loading file 'rr-empty-subnode.alt'.
```


hal-00349898, version 1 - 5 Jan 2009

hal-00349898, version 1 - 5 Jan 2009

```
arc>to_lustre Main
node EmptyNode(noinput : bool)
    returns (noreturn : bool);
let
   noreturn = true;
tel

node Main(noinput : bool)
    returns (noreturn : bool);
var
   a1_noreturn, a2_noreturn, a3_noreturn : bool;
let
   noreturn = true;
   a3_noreturn = EmptyNode(true);
   a2_noreturn = EmptyNode(true);
   a1_noreturn = EmptyNode(true);
tel
arc>
```

Customization: The user can customize the behaviours of translators. Configuration options are listed in the appendix A.

5 AltaRica to Lustre

5.1 Restrictions on AltaRica models

Except for signatures, any AltaRica model can be translated into Lustre but, the resulting program is not necessarily directly usable with the Lustre tool set. In order to get the best result the user has to respect some writing rules:

Orientation attributes: In the genuine AltaRica model[4, 1], flows are not oriented i.e. nodes have neither inputs nor outputs, but only flows. The language allows the specification of *attributes* (which are simple identifiers) for each variable or event. *Attributes* do not actually change the AltaRica semantics (except for visibility); they are annotations interpreted by tools. Two attributes, `in` and `out`, are used by the translator to change the default translation method of variables. If the attribute `in` (resp. `out`) is attached with a flow variable then this variable is translated has an input (resp. output) variable of the resulting Lustre node. This aspect of the translation is detailed in sections 5.2.1 and 5.3.2. A flow variable with no orientation attribute is translated has an output variable.

Non-Determinism: This is an important feature of the AltaRica language which allows to abstract behaviours of the modelled systems. Unfortunately such feature is not suitable for a programming language like Lustre which rejects non-determinism. In the case of a non-deterministic model, the translator arbitrarily enforces the choice in non-deterministic states (see section 5.2.2).

Priorities: This feature of the language is not handled, mainly because it does not exist in the Lustre language. An AltaRica model containing priorities can be compiled but the priorities are ignored.

Broadcast vectors: are synchronization vectors where some events are marked with a `?` to indicate that they are not mandatory to the synchronization. These vectors induce implicit priorities which are not handled; thus, broadcast vectors are translated as classical synchronization vectors.

Signatures: The translator requires that signatures do not use compound types (i.e structures or array) as their arguments or return value.

Parameters: Parameters are local or global constants without a value. They can be used in any ALTARICA expression: an assertion, a guard, ... They may appear in type definitions (e.g. the bounds of a range) or dimension of subnode arrays. The translator does not support this latter kind of parameters and reject the model.

For example the following node `N1` can be translated while `N2` is not accepted by the translator.

```
node N1
  param N : integer;
  state x : integer;
  assert x < N
edon
```

```
node N2
  param N : integer;
  state x : [0,2*N]
  assert x < N
edon
```

Given an ALTARICA node *N*, only data depending on *N* are translated: global parameters, types, signatures and subnodes. The translation algorithm is applied recursively in a depth-first way on the subnodes of *N*.

In order to be accepted by the LUSTRE tools, the dot notation (`.`) used for ALTARICA identifiers is replaced by the underscore (_) character.

5.2 Translation of leaf nodes

5.2.1 I/O interface

Each LUSTRE node is defined over 3 sets of variables: *inputs*, *outputs* and *local* variables. These sets of variables receive:

- the translation of parameters;
- the translation of "data" variables (i.e. states and flows);
- the translation of events and synchronization vectors (see 5.3.2);
- and some other (local) auxiliary variables used to improve human readability.

If they are all translated into LUSTRE variables, parameters, variables and events, do not follow the same propagation rules along the hierarchy and thus, must be treated in different ways as presented in the sequel. The table 1 summarizes all this translations.

Parameters: The translation algorithm takes into account only not valued parameters; the others are replaced by their value. Parameters do not have visibility or orientation attributes. Each parameter is prefixed with the word `p_` and appears in the signature of a node as a `const` input.

Flow and state variables: In order to deal with visibility constraints we associate to each variable of LUSTRE nodes an ALTARICA visibility attribute. These attributes are not actually translated; their are used for the translation of hierarchical nodes (see 5.3.2).

Private (state or flow) variables are translated as local (i.e `var`) variables. Flow variables are translated using their orientation and visibility attributes; if no orientation attribute is specified then the variable is translated as an output variable (with the specified visibility attribute). Orientation attributes are ignored for state variables which are translated as output variables. Identifiers of flow (resp. state) variables are prefixed with the word `f_` (resp. `s_`).

We recall that the default visibilities of flow and state variables are, respectively, `parent` and `private`.

Events: The most intuitive way to translate ALTARICA events is as Boolean inputs but this choice does not define the mapping between events and dataflow semantics. We must define the meaning of changes between Boolean values of the LUSTRE variable and their correlation with the occurences of the ALTARICA event. Let *e* be an event:

hal-00349898, version 1 - 5 Jan 2009

hal-00349898, version 1 - 5 Jan 2009

- A first interpretation of the Boolean flow associated with *e* is shown on the figure 2.a: it means that *e* occurs on pulse-edges. This semantics respects the instantaneity of ALTARICA events but it is not easy to use or intuitive for both LUSTRE simulation and LUSTRE verification.
- The second choice (figure 2.b) is more intuitive since it maps the value `true` with occurences of the event: *e* is present on each rising edges. The main drawback of this semantics is that the flow must go back to `false` before the next occurence of *e*; thus, *e* can not occur at two consecutive (discrete) instants.
- The last choice (figure 2.c) is very simple and intuitive: *e* occurs when the flow is `true`. This choice has the drawback that *e* is not instantaneous and is present all along the cycle. This interpretation does not follow the ALTARICA semantics of events. However, if we consider the semantics of LUSTRE programs, it does not matter since nothing happens between two ticks of the basic clock. The translator implements this choice.

The variables used in place of events are suffixed with an underscore character (_). The ε event does not appear in the I/O interface of nodes. `private` attribute is ignored for events.

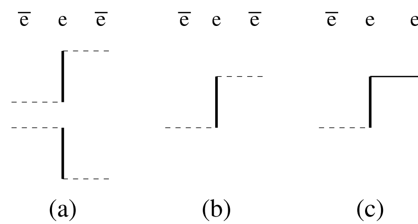


Figure 2: Event-flow semantics

LUSTRE does not allow empty input/output interface. In order to treat such a case we add two dummy Boolean variables `noinput` and `noreturn` if there is no input variable and/or no output variable. These variables are set to `true`.

Example 3 *In this example we show the use of dummy variables* `noinput` *and* `noreturn`*, and the result of the translation of* ALTARICA *variables, parameters or events. The reader can note the presence of two* `ec_x` *variables; these ones are auxiliary variables generated by the translation of transitions (addressed by the next section).*

ALTARICA

```
node EmptyNode
edon

node NodeWithEvents
  param N : bool;
  flow  i : bool : in;
        o : bool : out;
        l : bool : private;
  state s : bool;
        t : bool : public;
  init  s := true, t := true;
  event a, b;
  trans true |- a, b -> ;
edon
```

LUSTRE

```
node EmptyNode(noinput : bool)
    returns (noreturn : bool);
let
   noreturn = true;
tel

node NodeWithEvents__(const p_N : bool;
                      a_, b_, f_i : bool)
    returns (s_t, f_o : bool);
var
   f_l, s_s, ec_0, ec_1 : bool;
let
   ...
tel
```

hal-00349898, version 1 - 5 Jan 2009

AltaRica			Lustre ID	var. type
state X		private	s_X	local
		parent/public		out
not initialized state X			const INIT_VAL_X	in
flow X	in	private	f_X	local
		parent/public		in
	[out]	private		local
		parent/public		out
parameter X			const p_X	in
event X		private	X_	in
		parent/public		

Table 1: This table summarizes the translation of ALTARICA "variables" (i.e. states, flows, events and parameters) according to their orientation and visibility attributes.

5.2.2 Transitions

Enabling conditions: In order to improve readability the first work realized by the translator prior to the translation of transitions is the creation of a local Boolean variable for each transition. Each such variable represents the value of the enabling condition of its associated transition. These variables are named `ec_`*i* where i is some internal index used for the transition. If we suppose that the i^{th}transition is guarded with the Boolean expression G_i and labelled with the event e_i then the enabling condition variable `ec_`*i* is defined by: `ec_`*i* `= false -> pre(`G_i`) and` e_i.

The ε does not appear explicitly and thus, no `ec_`*i* is defined for it. The ALTARICA semantics specifies that the ε event is always possible; here we specify that ε occurs only when no other event is possible.

According to ALTARICA semantics, we have to ensure that:

1. at most one event is treated at each instant (including ε).
2. at least one transition is possible (including ε);

The first constraint is guaranteed using an assertion (which use '`#`' the *at most* operator); it expresses the mutual exclusion of input events. The second one is guaranteed by the generation of assertions which specify that if an input event occurs then at least one transition is enabled for this event. If no event occurs then ε is considered.

Generation of assignments: It would be more in the "spirit" of LUSTRE to generate one equation for each state variable but, in order to deal with some non-determinism problems (see below), variables are assigned by means of vectors of variables. All assignments are gathered in one vectorial expression consisting in nested `if-then-else` expressions where decision conditions are the variables `ec_`*i* and `then` expressions are assigned values. Of course, nested `if-then-else` operations are preceded by the initial values according to the ones specified in the ALTARICA description; if a state variable v is not initialized the translator adds an input constant `INIT_VAL_`*v* to the signature of the node.

Example 4 *The following example illustrates the translation of transitions.*

hal-00349898, version 1 - 5 Jan 2009

ALTARICA

```
node Valve
 event open, close, fail_stuck
 state
  is_open, stuck : bool;
  d : integer;
 init
  is_open := true,
  stuck := false;
 trans
  is_open and not stuck
    |- close -> is_open := false;
  not (is_open or stuck)
    |- open -> is_open := true;
  not stuck
    |- fail_stuck -> stuck := true;
edon
```

LUSTRE

```
node Valve(const INIT_VAL_d : int;
           open_, close_, fail_stuck_ : bool)
    returns (noreturn : bool);
var
   s_d : int;
   s_stuck, s_is_open, ec_0, ec_1, ec_2 : bool;
let
   noreturn = true;
   ec_0 = false ->
    pre (s_is_open and not s_stuck) and close_;
   ec_1 = false ->
    pre (not (s_is_open or s_stuck)) and open_;
   ec_2 = false ->
    pre (not s_stuck) and fail_stuck_;

   (s_is_open,s_stuck,s_d) =
     (true,false,INIT_VAL_d) ->
     if ec_2 then pre (s_is_open,true,s_d)
     else (if ec_1 then pre (true,s_stuck,s_d)
     else (if ec_0 then pre (false,s_stuck,s_d)
     else pre (s_is_open,s_stuck,s_d)));

   assert( close_ =>
           (false -> pre (s_is_open and
                          not s_stuck)) );
   assert( fail_stuck_ =>
           (false -> pre (not s_stuck)) );
   assert( open_ =>
           (false -> pre (not (s_is_open or
                          s_stuck))) );
   assert( #(open_,close_,fail_stuck_) );
tel
```

Dealing with non-determinism: ALTARICA allows the description of not deterministic models; unfortunately, LUSTRE tools reject such programs. The use of nested `if-then-else` with vectorial assignments resolves the non-determinism problem because this method enforces one choice among the possible non-deterministic actions.

The user has the choice of taking care or not of the determinism of its translated ALTARICA "program". If he is not interested by the states where non-determinism occurs, the user can customize the translator (see appendix A.2) in order to generate a new assertion which enforces the enabling conditions of transitions to be mutually exclusive. On the example 4, if we generate this assertion, the transator adds the following line: `assert( #(ec_0,ec_1,ec_2) )`. The figure 3 shows the effect of the assertion.

5.2.3 Assertions

To be valid all local and output variables of a LUSTRE node must be uniquely defined by an equation i.e. must be the left member of one and only one equation (the variable might however appear in any number of assertions or right members).

From both syntactical and semantical points of view, such constraint does not exist in the ALTARICA language. In order to ensure that LUSTRE compilers will not systematically reject translated models we can not translate ALTARICA assertions into LUSTRE `assert`. The translator tries to build as many equations it is possible to find (syntactically) in assertions. To achieve this goal this heuristic uses two passes:

1. In the first one, it explodes all assertions which are conjunctions and considers each operand of the `and` operator

hal-00349898, version 1 - 5 Jan 2009

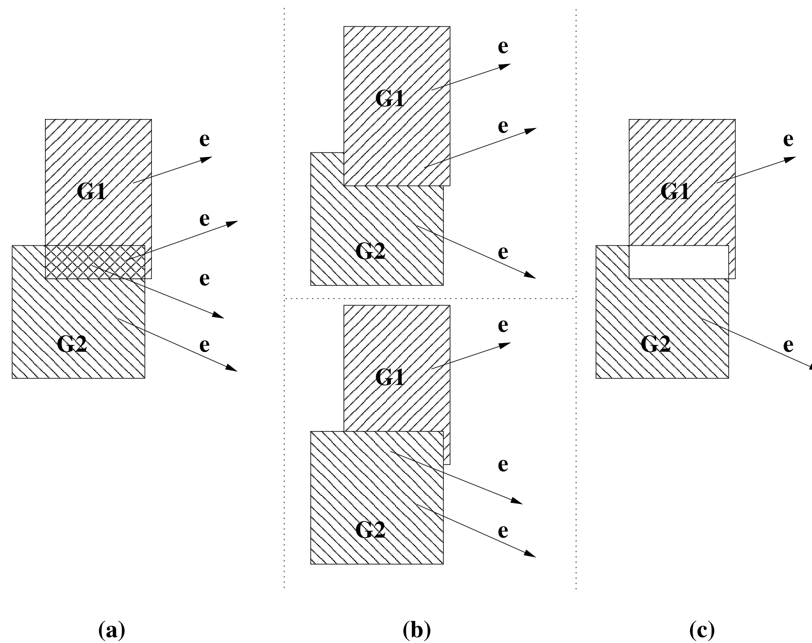


Figure 3: This figure presents the enabling state-spaces of two transitions guarded with non-exclusive condition G_1 and G_2. Possible outgoing edges on the occurence of the same event e are shown. In presence of non-determinism ALTARICA semantics allows all possible behaviours (figure **a**). By default, the translator choose one scenario (figure **b**) but if the assertion $\#(G_1, G_2)$ is activated, $G_1 \cap G_2$ is not authorized and we get the right most case (figure **c**).

as a new assertion.

2. In the second pass, for each assignable variable v in the node (i.e v is a local or output variable not already assigned), it looks for an assertion of the form v = *value* or *value* = v. If such an assertion is found, it is removed from assertions and translated as a LUSTRE equation. The pass stops after the first equation is found.

At the end of the translation, warning messages are displayed to inform the user if there exists variables with no definition.

Example 5 *The following example illustrates the translation of* ALTARICA *assertions. The first assertion is splitted into two sub-assertions,* `a > 10` *and* `b = 20`*; the second one is considered as a definition of the variable* `b`*. A warning message indicates that the output variable* `a` *is not defined which is only constrained by an assertion.*

ALTARICA	LUSTRE
`node Main`	`-- WARNING : output/local variable 'f_a' \`
`flow`	`in node 'Main' is not defined.`
`i : integer : in;`	`node Main(f_i : int)`
`a, b, c : integer : out;`	`returns (f_b, f_c, f_a : int);`
`assert`	`let`
`a > 10 and b = 20;`	`f_b = 20;`
`c = (if  a < 15 then i+1 else i-1);`	`f_c = if f_a < 15 then f_i+1 else f_i-1;`
`edon`	
	`assert( f_a > 10 );`
	`tel`

5.3 Translation of hierarchical nodes

The algorithm used to translate leaves has to be adapted for hierarchical nodes. The I/O interface has to be extended, transitions must be adapted to synchronized events and assertions have to handle subnode calls.

5.3.1 Hierarchy and subnode calls

The ALTARICA-TO-LUSTRE translator uses a depth-first method. Since two subnodes sn_1 and sn_2 of type *SN* can have different initial state specifications, the translator creates for each initial state specification a new translation of *SN* (the new version is suffixed with a number).

The ALTARICA hierarchy is translated by means of LUSTRE node calls. Due to the depth-first translation algorithm, subnodes are translated prior the embedding node; thus, the complete I/O interface of all subnodes is determined. For each subnode *sn*,

- Each variable appearing in its I/O interface is duplicated in the embedding node (see below for attributes); variables are prefixed with *sn_*.
- Then, an equation defining the vector of ouput variables of *sn* is created. The vector is set to the value of the call to *sn* applied to its input variables.

5.3.2 Subnode variables and synchronizations

Parameters: In order to make easier their valuation by the user, parameters are propagated along the hierarchy of LUSTRE nodes up to the toplevel node which gathers all existing parameters. Of course each parameter is prefixed with its associated subnode name and, appears in the signature of the embedding node as a `const` input.

Subnode inputs/outputs handling: The I/O interface of all subnodes is added to the embedding node sets of variables (but each variable is prefixed with the name of the subnode). The visibility attribute of this added variables is set according to the attribute of the original variable as follows (local variables of subnodes are not visible):

- `public` variables remain `public` and keep their orientation attribute;
- `parent` variables become local variables; the orientation attribute disappears.

Example 6 *The following example shows how the ALTARICA hierarchy is translated using subnode calls and how parameters and variables pass through the hierarchy. The I/O interface of each subnodes `a1`, `a2` and `b` is duplicated into the embedding node `Main`. The variables are translated into `Main` according to their visibility and orientation attributes:*

- *Since their visibility attribute is `parent`, the flow variable `i` and the state variable `s` of `SubNode` become local variables of `Main` (`a1_s_s`, `a1_f_i`, `a2_s_s` and `a2_f_i`).*
- *The parameter `N` of `AnotherSubnode` passes through the hierarchy and becomes a `const` input (`b_p_N`) of `Main`.*
- *Due to their (default) visibility attribute, the flow variables `i` and `o` of `AnotherSubnode` become local variables of `Main`.*
- *The `public` state variable `t` of `AnotherSubnode` becomes a public output (`b_s_t`) of `Main`.*

The reader can notice that, according to the initial value of its state variable `s`, the node `SubNode` has been translated twice from the `Main` node.

hal-00349898, version 1 - 5 Jan 2009

hal-00349898, version 1 - 5 Jan 2009

ALTARICA	LUSTRE

ALTARICA

```
node SubNode
  flow i : bool : in;
  state s : integer : parent;
  init s := 10;
edon

node AnotherSubNode
  param N : bool;
  flow  i : bool : in;
        o : bool : out;
        l : bool : private;
  state s : bool;
        t : bool : public;
  init  s := true, t := true;
edon

node Main
  sub a1, a2 : SubNode;
      b : AnotherSubNode
  init a1.s := 11;
edon
```

LUSTRE

```
node AnotherSubNode__(const p_N : bool;
                      f_i : bool)
   returns (s_t, f_o : bool);
var
   f_l, s_s : bool;
let
   (s_t,s_s) = (true,true) -> pre (s_t,s_s);
tel

node SubNode(f_i : bool)
   returns (s_s : int);
let
   s_s = 10 -> pre s_s;
tel

node SubNode_1(f_i : bool)
   returns (s_s : int);
let
   s_s = 11 -> pre s_s;
tel

node Main(const b_p_N : bool; noinput : bool)
   returns (b_s_t : bool);
var
 a1_s_s, a2_s_s : int;
 a1_f_i, a2_f_i, b_f_o, b_f_i : bool;
let
 (b_s_t,b_f_o) = AnotherSubNode__(b_p_N,b_f_i);
 a2_s_s = SubNode(a2_f_i);
 a1_s_s = SubNode_1(a1_f_i);
tel
```

Events: While for leaf nodes the event interface is built with their elementary events, the one for compound nodes is built with their synchronization vectors. For each implicit or explicit vector, a Boolean input variable is created. Then, each event input of subnodes is defined by the disjunction of vectors where they occur into.

For a compound node, the translation of transitions has to handle the synchronization constraint. As for subnode event inputs, in the enabling conditions (see 5.2.1) local events of the embedding node are replaced with the disjunction of vectors where it occurs into.

The name of the variable associated with a vector is the concatenation of its components; in the case of (implicit) vectors created for `public` events, the name of the event is not duplicated.

As a consequence of this algorithm, the toplevel node has as many input events as there are global events (i.e synchronization vectors) in the system.

Example 7 *The following example shows how synchronization vectors are translated into input variables.*

The synchronization constraint for the node `C` *contains two explicit vectors,* $\langle \varepsilon, a.e, b.e\rangle$ *and* $\langle \varepsilon, a.f, b.e\rangle$*, and two implicit ones,* $\langle \varepsilon, a.\varepsilon, b.f\rangle$ *and* $\langle \varepsilon, a.\varepsilon, b.\varepsilon\rangle$*. The last one is ignored; the three others correspond to the three inputs of the node. One can remark the disjunction of* $\langle a.f, b.e\rangle$ *(*`a_f__b_e_`*) and* $\langle a.e, b.e\rangle$ *(*`a_e__b_e_`*) used as the value of the event input* `e_` *for the call to the subnode* `B`*.*

The synchronization constraint for the `Main` *node is* $\langle m, C.\varepsilon\rangle$ *and* $\langle \varepsilon, C.\varepsilon\rangle$ *which generates the seven vectors (epsilons are removed):* $\langle m, a.e, b.e\rangle$*,* $\langle m, a.f, b.e\rangle$*,* $\langle m, b.f\rangle$*,* $\langle m\rangle$*,* $\langle a.e, b.e\rangle$*,* $\langle a.f, b.e\rangle$ *and* $\langle b.f\rangle$*. Each vector generates one input variable.*

The reader should note that the enabling conditions are adapted to take into account synchronization vectors. For example in the `Main` *node, the condition* `ec_0` *which enables the transition when the event* `m` *occurs is guarded by*

the disjunction of vectors synchronizing m.

ALTARICA

```
node A
  event e, f;
  trans true  |- e, f ->
edon

node B
  event e, f;
  trans true  |- e,f ->
edon

node C
  sub a : A;
      b : B;
  sync <a.e,b.e>;
       <a.f,b.e>;
edon

node Main
  event m;
  state s : bool;
  init s := true;
  trans s |- m -> s := not s;
  sub
      c : C;
edon
```

LUSTRE

```
node B(f_, e_ : bool) returns (noreturn : bool);
let ... tel;

node A(f_, e_ : bool) returns (noreturn : bool);
let ... tel;

node C(b_f_, a_f__b_e_, a_e__b_e_ : bool)
    returns (noreturn : bool);
var
   a_noreturn, b_noreturn : bool;
let
   noreturn = true;
   b_noreturn = B(b_f_,a_f__b_e_ or a_e__b_e_);
   a_noreturn = A(a_f__b_e_,a_e__b_e_);

   assert( #(a_e__b_e_,a_f__b_e_,b_f_) );
tel

node Main(m_, c_b_f_, m__c_b_f_, c_a_f__b_e_, c_a_e__b_e_,
          m__c_a_f__b_e_, m__c_a_e__b_e_ : bool)
    returns (noreturn : bool);
var
   ec_0, s_s, c_noreturn : bool;
let
   noreturn = true;
   c_noreturn = C(c_b_f_ or m__c_b_f_,
                  c_a_f__b_e_ or m__c_a_f__b_e_,
                  c_a_e__b_e_ or m__c_a_e__b_e_);
   ec_0 = false -> pre s_s and (m_ or
                                m__c_b_f_ or
                                m__c_a_f__b_e_ or
                                m__c_a_e__b_e_);
   s_s = true ->
    if ec_0 then pre (not s_s) else pre s_s;

   assert( (m_ or m__c_b_f_ or m__c_a_f__b_e_ or
            m__c_a_e__b_e_) => (false -> pre s_s) );
   assert( #(c_a_e__b_e_,c_a_f__b_e_,c_b_f_,m__c_a_e__b_e_,
             m__c_a_f__b_e_,m__c_b_f_,m_) );
tel
```

5.4 Domains, signatures and global constants

5.4.1 Domains

The LUSTRE language has 3 basic types, `bool`, `int` and `real` and allows the declaration of *external* types. The translation of ALTARICA integers (`integer`), Booleans (`bool`) and abstract domains (`sort` X) is, respectively, `int`, `bool`, and `type` X. The translation of other domains is made as follows:

Enumerations are sets of names belonging to a same namespace. Since one can associate to each name a unique number, enumerations are simply translated into `int`. For each such name n required by the translation we create a global integer constant `ENUM_`n (the prefix can be customized).

hal-00349898, version 1 - 5 Jan 2009

For each variable x taken its value into an enumeration domain, we add the corresponding constraint as an assertion of the LUSTRE node; this constraint is simply the disjunction of all possible assignments of x.

Arrays and Structures are exploded using a depth-first algorithm which creates as many components as there is structure fields or array cells. If the translated domain concerns a constant (resp. a variable) then a constant (resp. a variable) is created for each component of the compound domain.

The example 8 page 18 presents the translation of various domains.

5.4.2 Signatures

LUSTRE allows the declaration of *external* functions. This allows us to translate ALTARICA `signatures` as `functions` in a straightforward manner. We use a simple numbering pattern to respect the naming requirement of parameters and return values. Only scalar types are supported for arguments and the return value.

The example 8 below shows instances of translated signatures.

5.4.3 Global constants

ALTARICA and LUSTRE allow the declaration of constants which are valued or not; in the latter case they are called *parameters*. In order to "optimize" models, the value of constants is used as soon as possible. Thus, only constants with no value (i.e. parameters) are translated to LUSTRE. Except for its domain, the translation of a constant is direct. In the case of an enumeration domain, we assume that the use of the constant will constraint the value of the corresponding LUSTRE constant.

The following example shows several translations of global constants.

Example 8 *The following (not realistic) example shows the* LUSTRE *code produced by the translation of domains, constants and signatures.*

Some remarks about translated domains:

- *The enumeration domain* `q_status` *does not appear in the result since it is replaced by* `int` *but global constants* `ENUM_EMPTY`*,* `ENUM_FULL` *and* `ENUM_UNKNOWN` *are created in order to constraint variables; this is the case of the field* `qs` *(translated as* `f_f_qs`*) of the flow variable* `f`*.*
- *One can notice the splitting of the compound domain of the variable* `f` *(as well as its field* `b`*).*

One can also notice that `N` *is splitted into two Boolean constants. Despite the only use of* `N[1]`*, both components are generated. However the translator tries to generate only useful data; for example, the unused signature* `q_get_top` *or the global constant* `B` *are not translated.*

hal-00349898, version 1 - 5 Jan 2009

hal-00349898, version 1 - 5 Jan 2009

ALTARICA

```
domain q_status =
         { EMPTY, FULL, UNKNOWN };

sort queue, object;

sig q_get_size : queue -> integer;
sig q_is_empty : queue -> bool;
sig q_concat : queue * queue -> queue;
sig q_get_top : queue -> object;
sig q_get_status : queue -> q_status;

const EMPTY_QUEUE : queue;
const N : integer[2];
const B : bool[3];

node Main
  flow
    f : struct
         q : queue;
         qs : q_status;
         r : [0,10];
         b : bool[3];
        tcurts : in
  assert
    (q_get_size(f.q) > 0) =
              not q_is_empty(f.q);
    q_is_empty(q_concat(EMPTY_QUEUE,
                        EMPTY_QUEUE));
    q_is_empty(f.q) =
        (q_get_status(f.q) = EMPTY);
    f.r < N[1]
edon
```

LUSTRE

```
const ENUM_EMPTY = 0;
const ENUM_FULL = 1;
const ENUM_UNKNOWN = 2;

type        queue;

function q_get_size
        (q_get_size_arg_0 : queue)
        returns (q_get_size_return : int);

function q_is_empty
        (q_is_empty_arg_0 : queue)
        returns (q_is_empty_return : bool);

function q_concat
        (q_concat_arg_0 : queue;
         q_concat_arg_1 : queue)
        returns (q_concat_return : queue);

function q_get_status
        (q_get_status_arg_0 : queue)
        returns (q_get_status_return : int);


const EMPTY_QUEUE : queue;
const N_0_ : int;
const N_1_ : int;

node Main(f_f_r, f_f_qs : int;
          f_f_q : queue;
          f_f_b_0_, f_f_b_2_,
          f_f_b_1_ : bool)
        returns (noreturn : bool);
let
 noreturn = true;

 assert( f_f_qs = ENUM_UNKNOWN or
         f_f_qs = ENUM_FULL or
         f_f_qs = ENUM_EMPTY );
 assert( 0 <= f_f_r and f_f_r <= 10 );
 assert( (q_get_size(f_f_q) > 0) =
          (not q_is_empty(f_f_q)) );
 assert( q_is_empty(q_concat(EMPTY_QUEUE,
                             EMPTY_QUEUE)) );
 assert( q_is_empty(f_f_q) =
        (q_get_status(f_f_q) = ENUM_EMPTY));
 assert( f_f_r < N_1_ );
tel
```

5.5 Properties of the translator

Two properties on ALTARICA models are important:

Well oriented: For a given valuation of state variables, the number of solutions for flow variables may be O (the post-condition is apply), 1 (the model is flow deterministic) or more than one (the model is not flow deterministic).

Generally, there is more than one solutions, and a good orientation of the model is a partition of the flow variables such that for any valuation of states and `in` flow variables, there is only one solution for `out` flow variables. To have a well oriented model, it is sufficient that (i) all `out` flow variables are defined by an equation, (ii) there is not a cycle dependencies between `out` flow variables. These two conditions are tested by the translator and warning messages are displayed to inform the user there exists variables with no definition.

State deterministic: For a given event, the number of solutions for assignment of state variables may be 0 (the post-condition is apply), 1 (the model is state deterministic) or more than one (the model is not state deterministic). Generally, there is only one solution, and if there is more than one, the designer must have explicitly write them. Nevertheless, the translator don't display a warning for this property since it is a difficult task without computing the explicit semantics of the model.

Properties of the LUSTRE program obtain by the translator depends on properties of the given ALTARICA model.

Not well oriented: The translator display a warning, and the resulting program is not accept by LUSTRE tools.

Well oriented and state deterministic: There is no warning, and the resulting program is accept by LUSTRE tools.

Well oriented and not state deterministic: There is no warning, and the result depends on the `deterministic` option.

deterministic=true: The resulting program is accept by LUSTRE tools, but a violation assertion appears when the non deterministic choice appears.

deterministic=false: The resulting program is accept by LUSTRE tools. There is no violation assertion since the result is a deterministic implementation of the non deterministic model.

6 LUSTRE to ALTARICA

A LUSTRE program is a set of abstract `types` and `functions`, abstract or concrete `constants` and `nodes`. The translation consists in two parts: the first one translates in a straightforward way `types`, `functions` and `constants`; the second one compiles `nodes` in a depth-first manner.

Each node encodes a function which uses other nodes as sub-functions; no recursive call is allowed. Given a root node N, its translation into the ALTARICA language consists in:

- the translation of each sub-functions used by N;
- the translation of N itself.

In the sequel we denote by N the LUSTRE node under translation and N_a its associated ALTARICA node.

6.1 Restrictions on Lustre programs

The LUSTRE-TO-ALTARICA translator supports only a subset of the LUSTRE language:

- Only the basic clock is authorized.
- The `current` and `when` operators are restricted to the form `current( X when Y )`.
- Arrays are not supported.

hal-00349898, version 1 - 5 Jan 2009

hal-00349898, version 1 - 5 Jan 2009

6.2 Types, functions and constants

LUSTRE supports 3 native types: integer values (`int`), Booleans (`bool`) and real numbers (`real`). Since Booleans are also a basic type of ALTARICA, the translation of `bool` is straightforward.

In order to do *model-checking*, the `arc` tool requires that the variables take their values into finite domains. The user has the choice (see appendix A.1) to translate the `integers` as the `integer` ALTARICA type or as a fixed finite range `[min,max]` (`min` and `max` are chosen by the user).

The `real` numbers are not supported by the ALTARICA language. We use the abstract types and signatures of the language to translate them (see below).

LUSTRE also allows the declaration of *external* types of data. These types are simply translated using the abstract type specifier `sort` with the name used in the LUSTRE program.

The LUSTRE `functions` are translated using the ALTARICA signatures (used to specify abstract functions). The identifiers used to name the parameters and the return values of the functions are lost by the translation.

The `constants` declared at the toplevel of the LUSTRE program are translated as global ALTARICA constants. Not valued constants are translated as parameters.

Example 9 *The following `arc` session shows how abstract types and functions are translated:*

```
$ cat rr-abstract-types.lus
type time, object;

function get_timestamp(obj : object) returns (ts : time);

const NULL_OBJECT : object;
const NULL_OBJECT_TIMESTAMP = get_timestamp(NULL_OBJECT);

node Main(t : time; o : object) returns (op : object);
let
 ...
tel.

$ arc rr-abstract-types.lus
Loading file 'rr-abstract-types.lus'.
L2A: translating lustre node 'Main'.
arc>dump Main
sort object;
sort time;

sig get_timestamp : object -> time;

const NULL_OBJECT : object;

node Main
...
```

The constant `NULL_OBJECT_TIMESTAMP` does not appear in the ALTARICA *specification because it has been replaced by its definition `get_timestamp(NULL_OBJECT)`.*

Translation of `reals`: `real` is one of the native type of the LUSTRE language. This type is not supported by ALTARICA but it can be translated using abstract data types and signatures[1].

The type `real` is translated by a new (abstract) type `REALS`. Each time a `real` constant (e.g. `3.1415`) is used in an expression, an abstract constant is created (e.g. `'3.1415'`) and is used as the translation of the LUSTRE constant. The zero value may appear with the translation of the `pre` operator (see 6.5); in this case, the constant is called `real_zero`.

[1] The user can customize the identifiers used for the translation (see appendix A.1); here we use the default values of these identifiers.

The translation of `real` related operations is done on demand. The translation of arithmetic operations and comparators is presented in the table 2.

LUSTRE Op.	ALTARICA signature
- (unary)	`sig real_neg : REALS -> REALS`
+	`sig real_add : REALS * REALS -> REALS`
-	`sig real_sub : REALS * REALS -> REALS`
*	`sig real_mul : REALS * REALS -> REALS`
/	`sig real_div : REALS * REALS -> REALS`
mod	`sig real_mod : REALS * REALS -> REALS`
<	`sig real_lt : REALS * REALS -> bool`
<=	`sig real_leq : REALS * REALS -> bool`
>	`sig real_gt : REALS * REALS -> bool`
>=	`sig real_geq : REALS * REALS -> bool`

Table 2: Translation table of arithmetic operators and comparators over real numbers.

Example 10 *The following* LUSTRE *code (on the left side below) models a division operation over real numbers. The* `divide` *node returns the quotient and the remainder of* `a` *divided by* `b`*. The translation of the* `divide` *node, its related operations and types is shown on the right column of the following table.*

LUSTRE

```
function floor(x : real)
   returns(y:real);

node divide(a, b : real)
   returns(q, r : real);
var
  d : real;
let
  d = if b <> 0.0 then b else 1.0;
  q = floor(a/d);
  r = a-d*q;
tel.
```

ALTARICA

```
sort REALS;

sig real_div : REALS * REALS -> REALS;
sig floor : REALS -> REALS;
sig real_mul : REALS * REALS -> REALS;
sig real_sub : REALS * REALS -> REALS;

const real_zero : REALS;
const '1' : REALS;

node divide
  flow
   /* 1 */ q : REALS : out;
   /* 2 */ r : REALS : out;
   /* 3 */ a : REALS : in;
   /* 4 */ b : REALS : in;
   /* 5 */ d : REALS : private;
  assert
   d=(if b!=real_zero then b else '1');
   q=floor(real_div(a,d));
   r=real_sub(a,real_mul(d,q));
  ...
edon
```

The reader should note the creation of the `real_zero` *constant. This constant is used by the translator to initialize (if necessary)* `real` *variables which are memorized using the* `pre` *operator (see 6.5).*

6.3 Variables, local variables and parameters

The translation of data-flow variables is straightforward using ALTARICA attributes:

- Input variables or parameters (i.e. input variables with the `const` specifier) are translated as flow variables with the default visibility and the `in` attribute.

hal-00349898, version 1 - 5 Jan 2009

- Output variables are translated as flow variables with the default visibility and the `out` attribute.
- Local variables are translated as private flow variables.

Example 11 *The following table shows on the left column the I/O interface of a* LUSTRE *node and, on the right column, its translation into* ALTARICA*:*

LUSTRE	ALTARICA

`node Main(const N : int; b : bool)`
`returns (o: int);`
`let`
`...`
	`node Main`
	`flow`
	`/* 1 */ o : integer : out;`
	`/* 2 */ N : integer : in;`
	`/* 3 */ b : bool : in;`
	`...`

6.4 Translation of subnode calls

A LUSTRE node can call other nodes as functions. Each node call is translated as a subnode of the caller. The parameter passing is realized using ALTARICA assertions which constrain the flow variables of subnodes.

Assume that $n : (i_1, \ldots, i_k) \rightarrow (o_1, \ldots, o_r)$ is a LUSTRE node with k inputs i_j and r outputs o_j, and that a_i are expression for $i = 1 \ldots k$. Each time a subnode call $n(a_1, \ldots, a_k)$ is encountered in an expression (i.e an equation or an assertion) of the node under translation N then:

- The node n is translated into n_a (a table is used to store already translated nodes).
- A new sub-node `sc_`m of type n_a is created in N_a. m is an integer ensuring unicity of the symbol `sc_`m.
- k constraints are added to the node N_a : `sc_`m`.`i_1 `=` $a_1, \ldots,$ `sc_`m`.`i_k `=` a_k.
- The expression $n(a_1, \ldots, a_k)$ is replaced by :
  - `sc_`m`.`o_1 if $r = 1$ (i.e. the return type of n is a scalar value and o_1 is its unique output variable).
  - the structure $\{$ `._field_0 = sc_`m`.`o_1`, ..., ._field_`$(r-1)$ `= sc_`m`.`o_r $\}$ if $r \geq 2$.

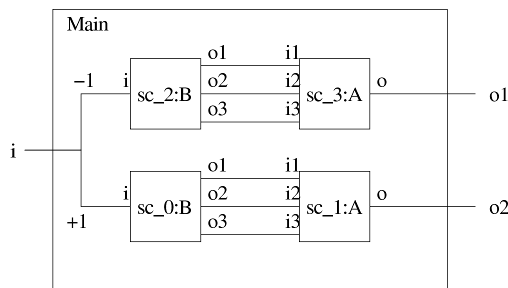
Example 12 *Consider the following* LUSTRE *program:*

```
node A(i1 : int; i2 : int; i3:int) returns (o: bool); let ... tel;

node B(i : int) returns (o1 : int; o2 : int; o3:int); let ... tel;

node Main(i : int) returns (o1, o2 : bool);
let
  (o1,o2) = (A(B(i+1)),A(B(i-1)));
tel;
```

The equation defining the local variables `o1` *and* `o2` *is translated as follows.*

hal-00349898, version 1 - 5 Jan 2009

```
node Main
 ...
  sub
   sc_3 : A;
   sc_2 : B;
   sc_1 : A;
   sc_0 : B;
  assert
   sc_0.i=i+1;
   sc_1.i1=sc_0.o1;
   sc_1.i2=sc_0.o2;
   sc_1.i3=sc_0.o3;
   sc_2.i=i-1;
   sc_3.i1=sc_2.o1;
   sc_3.i2=sc_2.o2;
   sc_3.i3=sc_2.o3;
   {._field_0 = o1, ._field_1 = o2}={._field_0 = sc_1.o, ._field_1 = sc_3.o};
 ...
edon
```



The last equality correspond to the vectorial equation defining the couple `(o1,o2)` *of the* LUSTRE *program. The other assertions model the parameter passing for the node calls; for instance, the first assertion* `sc_0.i = i+1` *corresponds to the call* `B(i+1)`*. All these assertions are represented on the above schema.*

6.5 Translation of the `pre` operator

The `pre` operator returns the value of its argument at the previous instant; it works as a memory of its argument. This operator does not exist in ALTARICA; it is translated using a state variable memorizing the current value of the argument of the operator.

If e is a LUSTRE expression, `pre(`e`)` is translated recursively as follows:

- if e is a vector $(e_1, \ldots, e_n)$ then the result is the translation of $(\mathtt{pre}(e_1), \ldots, \mathtt{pre}(e_n))$.
- else, if e is not a variable then we add to the ALTARICA node a new flow variable f_e[2] with the same type than e; the visibility of f_e is set to `private`. Then we add the assertion $f_e = e$ and we translate `pre(`f_e`)`.
- else, e is a variable and we add to the ALTARICA node a state variable `pre_`e with the same type than e. For native types, an initial value is assigned to `pre_`e: `false` for Booleans, `0` for `integers` and the abstract constant `real_zero` for `reals` (see the paragraph about the translation of `reals` page 21). Finally, each transition that will be created for the node N_a should assign the variable e to the variable `pre_`e.

Example 13 *The following example illustrates the translation of the operator* `pre`*. A flow variable* `_f_tmp_0` *is created to handle the value of the expression* `2*y+1`*. The memory is created using the state variable* `pre__f_tmp_0`*.*

The reader should note that `func` *is not a valid* LUSTRE *node because* `y` *has no initial value. The undefined value (called* nil*) does not exist in* ALTARICA *and, despite the* LUSTRE *semantics, the translator tries to initialize all translated variables.*

[2] The actual name of these variables is `_f_tmp_`n where n is an integer. This name might be customized; see A.1.

hal-00349898, version 1 - 5 Jan 2009

LUSTRE	ALTARICA

`node func(x : int)`
`   returns (y:int);`
`let`
`   y = pre(2*y+1)+x;`
`tel.`
	`node func`
	`  flow`
	`   /* 1 */ y : integer : out;`
	`   /* 2 */ x : integer : in;`
	`   /* 3 */ _f_tmp_0 : integer : private;`
	`  state`
	`   /* 1 */ pre__f_tmp_0 : integer;`
	`  event`
	`   /* 1 */ clock;`
	`  trans`
	`   true |- clock -> pre__f_tmp_0 := _f_tmp_0;`
	`  assert`
	`   _f_tmp_0=2*y+1;`
	`   y=pre__f_tmp_0+x;`
	`  init`
	`   pre__f_tmp_0 := 0;`
	`edon`

6.6 Translation of the "followed by" operator (->)

A LUSTRE expression defines a clocked sequence of values. The operator `->` (read "followed by") allows to define the initial value of such sequence.

The LUSTRE-TO-ALTARICA translator supports only one clock, the so-called *basic clock*, and faster clocks are not allowed (see 6.7). Since all flows are based on the same clock, expressions of the form E `->` F are translated by `if _is_init then` E `else` F where `_is_init` is a Boolean state variable added to the ALTARICA node N_a. This variable allows to check whether or not the node is in its initial state. The variable is created only if `->` is actually used.

Example 14 *The following example shows the creation of the* `_is_init` *state variable and its use to set the initial value of the flow variable* `y`*. The reader should note that the translation is not optimized: the definition of* `y` *is equivalent to* `0 -> x` *but the translator generates the three* `if-then-else` *operators used to define* `y`*.*

LUSTRE	ALTARICA

`node followed_by(x : int)`
`   returns (y:int);`
`let`
`  y = 0->1->2->x;`
`tel.`
	`node followed_by`
	`  flow`
	`   /* 1 */ y : integer : out;`
	`   /* 2 */ x : integer : in;`
	`  state`
	`   /* 1 */ _is_init : bool;`
	`  event`
	`   /* 1 */ clock;`
	`  trans`
	`   _is_init |- clock -> _is_init := false;`
	`   not _is_init |- clock -> ;`
	`  assert`
	`   y=(if _is_init`
	`        then 0`
	`        else if _is_init`
	`               then 1`
	`               else if _is_init then 2 else x);`
	`  init`
	`   _is_init := true;`
	`edon`

hal-00349898, version 1 - 5 Jan 2009

6.7 Translation of the `current`/`when` operators

The operator `when` is used to produce a data flow based on a clock slower than the basic clock. `current` allows to produce a data flow faster than its argument (but never faster than the basic clock). The LUSTRE-TO-ALTARICA translator supports only the basic clock; thus, the operators `when` and `current` are accepted for the translation only under the following syntactic restriction : `current(`E `when` F`)` where E and F are LUSTRE expressions.

Any expression `current(`E `when` F`)` is translated by a newly created LUSTRE variable `_f_tmp_`n (having the same type than E). This variable is then defined by the equation: `_f_tmp_`n `= if` F `then` E `else pre(_f_tmp_`n`)`

Example 15 *An example of `current/when` translation. Following the* LUSTRE *semantics of `when`, the variable `Y` is not defined (i.e. equal to* nil*) until `B` becomes `true`. The* ALTARICA *node behaves differently since any `pre_x` state variable is initialized.*

LUSTRE

```
node func(B : bool; X : int)
    returns (Y : int);
let
   Y = current(X when B);
tel.
```

ALTARICA

```
node func
  flow
   /* 1 */ Y : integer : out;
   /* 2 */ B : bool : in;
   /* 3 */ X : integer : in;
   /* 4 */ _f_tmp_0 : integer : private;
  state
   /* 1 */ pre__f_tmp_0 : integer;
  event
   /* 1 */ clock;
  trans
   true |- clock -> pre__f_tmp_0 := _f_tmp_0;
  assert
   _f_tmp_0=(if B then X else pre__f_tmp_0);
   Y=_f_tmp_0;
  init
   pre__f_tmp_0 := 0;
edon
```

6.8 Modelling and synchronization of clocks

The LUSTRE-TO-ALTARICA translator treats LUSTRE programs which are cadenced by one global clock. The *ticks* of this clock represent the discret instants where the values of data-flows are instantaneously and synchronously updated. In order to model the *ticks*, each ALTARICA node created by the translator is equipped with an event `clock`.

Transitions: Depending on the existence of the `_is_init` state variable (due to the translation of `->` operators – see 6.6) one or two transitions are created:

- If `_is_init` does not exist then we generate the transition `true |- clock ->`.
- else if it exists, we generate the two transitions:
  - `_is_init |- clock -> _is_init := false`
  - `not _is_init |- clock ->`.

Of course the assignment part of this generated transition(s) is extended with the assignments generated by the translation of the `pre` operators.

hal-00349898, version 1 - 5 Jan 2009

hal-00349898, version 1 - 5 Jan 2009

Synchronization: In order to ensure the synchronism of nodes, all `clock` events are synchronized at each level of the hierarchy.

Example 16 *If we go back to the example 12 (page 23), the* ALTARICA *node* `Main` *has to synchronize its 4 subnodes generated by calls* `A(B(i-1))` *and* `A(B(i+1))`*. The text produced by the translator is the following:*

```
node Main
  flow
   ...
  event
   /* 1 */ clock;
  sub
   sc_3 : A;
   sc_2 : B;
   sc_1 : A;
   sc_0 : B;
  trans
   true |- clock -> ;
  assert
   ...
  sync
   <sc_0.clock, clock, sc_1.clock, sc_2.clock, sc_3.clock>;
edon
```

7 Conclusion

The translators are effectives and the compilation process seems correct. They have to be used with real models in order to test the readability of the result.

Possibles improvements concern priorities and broadcast vectors.

References

[1] A. Arnold, , A. Griffault, and G. Point A. Rauzy. The AltaRica formalism for describing concurrent systems. *Fundam. Inf.*, 40(2-3):109–124, 1999.

[2] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[3] N. Halbwachs. A Synchronous Language at Work: the Story of Lustre. In *Third ACM-IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'2005*, Verona, Italy, July 2005.

[4] G. Point. *AltaRica: Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement*. PhD thesis, Universoté Bordeaux I, Talence, January 2000.

[5] G. Point and A. Rauzy. AltaRica - Constraint automata as a description language. *European Journal on Automation*, 1999. Special issue on the *Modelling of Reactive Systems*.

A Customization of the translations

The ARC tool uses a configuration file called `.arcrc` which is created in the home directory of the user. Each line of this file has the form *option-name* = *value*. The `.arcrc` file permits the user to configure translation-related data as presented in the following sections.

hal-00349898, version 1 - 5 Jan 2009

A.1 Customization of the Lustre to AltaRica Translation

Each configuration names related to the LUSTRE-TO-ALTARICA translator are prefixed by `translators.l2a.` (for example `translators.l2a.verbose`).

Name	Default	Comment
`verbose`	true	Allows the translator to display lots of information/debugging messages during the translation process.
`warning`	true	Requests the translator to display messages when some translation exceptions occur.
`integers.as_range`	false	If this option is set to `true` then the LUSTRE `integers` are translated as a range; its bounds are defined with the following options.
`integers.min`	-10	In the case where integers are represented by a range, this option defines its lower bound.
`integers.max`	10	In the case where integers are represented by a range, this option defines its upper bound.
`initial_state_name`	_is_init	The name of the Boolean state variable used to model the initial instant (see section 6.6).
`clock_event_name`	clock	The name of the event used to synchronize all LUSTRE nodes on the same clock (see section 6.8).
`subnodes_prefix`	sc_	The prefix used for the creation of ALTARICA subnodes replacing LUSTRE subnode calls (see section 6.4).
`pre_var_prefix`	pre_	The prefix used for the creation of variable implied into the translation of the `pre(.)` operator (see section 6.5).
`tmp_flow_var_prefix`	_f_tmp_	The prefix used for the creation of auxiliary flow variables (see sections 6.7 or 6.5).
`struct_field_prefix`	_field_	The prefix used for the translation of vectorial equalities (see section 6.4).
`reals.typename`	REALS	The name of the abstract domain used to translate the LUSTRE type `real` (see section 6.2).
`reals.zero`	real_zero	In the case where `real` type is used, this option is the name of the abstract constant used to represent the 0.
`reals.add`	real_add	Name of the signature used to represent the addition over `reals`.
`reals.div`	real_div	Name of the signature used to represent the division over `reals`.
`reals.mod`	real_mod	Name of the signature used to represent the modulo over `reals`.
`reals.mul`	real_mul	Name of the signature used to represent the multiplication over `reals`.
`reals.sub`	real_sub	Name of the signature used to represent the substraction over `reals`.
`reals.neg`	real_neg	Name of the signature used to represent the opposite unary operation over `reals`.
`reals.lt`	real_lt	Name of the signature used to represent the *less than* comparator over `reals`.
`reals.leq`	real_leq	Name of the signature used to represent the *less or equal* comparator over `reals`.
`reals.gt`	real_gt	Name of the signature used to represent the *greater than* comparator over `reals`.
`reals.geq`	real_geq	Name of the signature used to represent the *greater or equal* comparator over `reals`.

hal-00349898, version 1 - 5 Jan 2009

A.2 Customization of the AltaRica to Lustre Translation

Each configuration names related to the ALTARICA-TO-LUSTRE translator are prefixed by `translators.a2l.` (for example `translators.a2l.verbose`).

Name	Default	Comment
`verbose`	true	Allows the translator to display lots of information/debugging messages during the translation process.
`warning`	true	Requests the translator to display messages when some translation exceptions occur.
`deterministic`	true	If this option is set to `true` then the translator adds an assertion to each LUSTRE node which ensures that enabling conditions are mutually exclusive (see section 5.2.2).
`enum_prefix`	ENUM_	The prefix of global constants used in place of enumeration names declared in ALTARICA domains (see section 5.4.1).
`parameter_prefix`	p_	Prefix added to each translated parameter.
`flow_var_prefix`	f_	Prefix added to each translated flow variable.
`state_var_prefix`	s_	Prefix added to each translated state variable.
`event_var_suffix`	_	Prefix added to each translated event.
`missing_init_prefix`	INIT_VAL_	The prefix used to name constant inputs appearing when a state variable is not initialized (see section 5.2.2).
`guard_var_prefix`	ec_	Prefix of each auxiliary variable used to model enabling conditions of transitions (see section 5.2.2).
`noreturn_name`	noreturn	Name of the variable used when a LUSTRE as no output flow (see section 5.2.1).
`noinput_name`	noinput	Name of the variable used when a LUSTRE as no input flow (see section 5.2.1).
`signature_arg_prefix`	_arg_	Name used to prefix arguments of translated signatures (see section 8).
`signature_ret_suffix`	_return	Name used to suffix the output flow of a translated signature (see section 8).
`show_sections`	true	If this option is set to `true` the translator adds comments used to separate some parts of the generated LUSTRE code (e.g. constants, functions, ...).