The Multi-Paradigm Synchronous Programming Language LEA*

Leszek Holenderski, Axel Poigné

GMD – Forschungszentrum Informationstechnik GmbH
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany
{holenderski,poigne}@gmd.de

Abstract

We present uniform translation rules for three synchronous programming languages LUSTRE, ESTEREL and ARGOS, to a common intermediate format called Boolean automata. Such a uniform translation allows to smoothly combine the languages into one multi-paradigm language for programming synchronous reactive systems.

1 Introduction

LEA is a multi-paradigm language for programming synchronous reactive systems [3]. It is obtained by integrating three existing synchronous programming languages: LUSTRE [4], ESTEREL [2] and ARGOS [5].

LEA has been implemented in Synchronie Workbench [7] (SWB) which is an integrated development environment for writing, compiling, testing, verifying, simulating, animating and generating code for three synchronous languages: ARGOS, ESTEREL and LUSTRE. Its purpose is to provide the user with a complete system in which all the stages of software/hardware development for reactive systems can be performed.

A unique feature of SWB is its ability to smoothly integrate modules written in all three languages. This is made possible by the fact that all the compilers translate source programs into a common intermediate format called Synchronous Components (SC). The SCs produced by the various compilers can then be linked into a single SC representing the finished application.

The user of SWB can benefit in two obvious ways from such an integration. First, one gets a multi-paradigm synchronous programming language in

*The work was partially funded by the Esprit BRA **SYRF**, "Synchronous Reactive Formalisms" (Esprit Project 22703)

which graphical Argos, imperative Esterel and declarative Lustre are smoothly combined together. Second, there is no need for having many separate tools developed specifically for the three languages. They all can be shared since simulation, code generation for target languages, formal verification, code optimization, causality checks, etc., are all performed on the SCs.

We call our integration *smooth* for two reasons. First, after compiling a source module to the intermediate format, the module looses its origin, in that knowledge of the module's source language is no longer needed. Module headers and import conventions are uniform across languages, so using a module becomes as simple as calling an external procedure in a more conventional language. Second, we do not need a separate language for describing the integration of separately compiled modules into a final reactive system. Instead, we use the languages themselves. This is how it works.

All the three languages, when considered separately, are equipped with the concept of importing a module (refinement in Argos, **run** statement in Esterel, and node's application in Lustre). Importing is defined as textual inclusion of the imported module in the place of the import clause, via a kind of macro-expansion mechanism. Unfortunately, such a concept of import cannot be directly employed across different languages since both the imported module, say M, and the module which imports M must be written in the same language.

However, if all the languages are compiled to the same intermediate format, the macro-expansion mechanism can easily be lifted from the level of source modules to the level of compiled modules. In other words, instead of resolving import clauses during compilation, as it has been done so far, they are resolved during a separate phase called linking. A linker gets a compiled module, the main module which describes the overall architecture of the final reactive system, and resolves all the import clauses, by including the intermediate code of imported modules in the place of the corresponding import clauses. Since our import protocol is uniform across the different languages, the main module itself, and all the modules it imports, can be written in any of the three languages. An imported module can import other modules, provided there are no cycles in the import relation.

In this paper we consider only pure LEA, in which one can manipulate Boolean data only. The intermediate format used for such a simplified version is called *Boolean automata.* We first present translation rules to Boolean automata, for all the languages considered separately, and then discuss adjustments needed for the smooth integration of the languages.

Our translation rules also allow to compile pure LEA directly to silicon, in a way similar to hardware translation of pure Esterel [1]. However, our approach is algebraic in that we have designed a formal notation, based on a concise set of operations, in which we can precisely describe our translation rules in a compositional (syntax-directed) way.

2 Synchronous model

A synchronous reactive system, say S, communicates with its environment via input and output signals. Internally, S may also use some auxiliary signals, called *local* signals. S may consist of several components run in parallel.

There are two basic ingredients in the synchronous model of parallel programs: *discrete time* and communication via *instantaneous broadcast*.

Time is *discrete* in that time scale, usually represented by non-negative real numbers, is perceived instead as enumerable number of non-overlapping segments called *instants of time*. A synchronous reactive system is a cyclic (possibly non-terminating) process. Each cycle, a so-called *reaction*, is performed in one instant of time. A reaction consists in fetching inputs, computing, and emitting outputs. In fact, fetching, computing and emitting does not need to be performed in this order. Any order is permitted since a reaction is assumed to be atomic. For this reason, reactions and instants are synonymous.

If S consists of several components, which are run in parallel, all components perform their reactions simultaneously, i.e., in the same instant. The components communicate via *instantaneous broadcast*: a non-local signal emitted by one component is perceived as present (in the same instant it was emitted) by all components.

3 Formalisation of the synchronous model

In what follows, we only consider so-called *pure* reactive systems where all signals are pure. A pure signal is either present or absent (true or false, on or off, up or down). A signal is present if it is either provided by the environment, in case of input signals, or emitted by S, in case of output and local signals. S may query any signal for presence or absence.

Execution of S consists of a (possibly infinite) sequence of reactions. Each reaction starts in some state, with some input signals being present, and terminates in some state, with some output signals being emitted during the transition from the start to terminal state.

In general, the synchronous reactive system S with inputs In and outputs Out can be modeled by a labeled transition system $(\Sigma, \sigma_0, \Lambda, \longmapsto)$ where Σ is a set of states, $\sigma_0 \in \Sigma$ is an initial state, Λ is a set of labels and $\longmapsto \subseteq \Sigma \times \Lambda \times \Sigma$ is a transition relation which describes all possible reactions of S. Labels have the form I/O where $I \subseteq \mathit{In}$ and $O \subseteq \mathit{Out}$. A reaction is depicted by $\sigma \xmapsto{I/O} \sigma'$ where σ is the start state, I is the set of inputs present in the reaction, O is the set of outputs emitted by S during the reaction, and σ' is the state in which the reaction terminates.

As usual, a computation of S is a (possibly infinite) sequence of transitions such that the start state of the first transition is the initial state of S and the start state of a later transition is the terminal state of the previous transition. A computation is depicted by

$$\sigma_0 \xmapsto{I_1/O_1} \sigma_1 \xmapsto{I_2/O_2} \sigma_2 \longmapsto \cdots$$

Assume $S = S_1||S_2$. It turns out that it is difficult to define $\underset{S}{\longmapsto}$ compositionally, i.e., in terms of $\underset{S_1}{\longmapsto}$ and $\underset{S_2}{\longmapsto}$ alone, due to the phenomenon known as *instantaneous feedback*. A simple solution, described in [2], consists in considering a slightly different transition system $(\Sigma, \sigma_0, \Lambda', \rightarrow)$ whose labels have the form E/E' where E and E' (so-called *events*) are subsets of the set of *all possible* signals. E/E' denotes the fact that signals E' are emitted under the assumption that all the signals present during the reaction are exactly those in E. In order for a transition to specify a valid reaction, the condition $E' \subseteq E$ must hold. This restriction captures the instantaneous broadcast of signals: each signal emitted during a reaction is assumed to be present in the reaction.

Under such interpretation, the following simple rule captures the essence of $S_1||S_2$ (modulo termination of S):

$$\frac{\sigma_1 \xrightarrow[S_1]{E/E_1} \sigma_1' \quad \sigma_2 \xrightarrow[S_2]{E/E_2} \sigma_2'}{(\sigma_1, \sigma_2) \xrightarrow[S_1||S_2]{E/E_1 \cup E_2} (\sigma_1', \sigma_2')}$$

The relation $\longmapsto$ can then be defined in terms of $\rightarrow$, as

$$\sigma \xmapsto{I/O} \sigma' \quad \text{iff} \quad \sigma \xrightarrow{I \cup O/O} \sigma'$$

4 Boolean automata

Boolean automata specify pure synchronous reactive programs. They are compact representations of the labeled transition systems described in section 3, and serve as the common intermediate format to which pure LEA is translated.

A Boolean automaton, say $\mathcal{A}$, is simply a set of Boolean clauses of the form $v := b$ where $v \in \mathcal{V}$ and $b \in \mathcal{B}$. $\mathcal{V}$ denotes a fixed set of propositional variables and $\mathcal{B}$ denotes the set of all Boolean terms over $\mathcal{V}$. (In what follows, we use *ff* and *tt* to denote Boolean constants, and $\wedge$, $\vee$, $\neg$ to denote Boolean operators.)

As usual, any subset $V \subseteq \mathcal{V}$ can be regarded as a valuation of propositional variables, and we use $V(b)$ to denote the value of b under V.

Clause $v := b$ is said to *define* v and b is called a term defining v. We assume that each v is defined by at most one clause. For valuation V, if $V(b) = tt$ then the clause is said to *set* v.

$\mathcal{A}$ represents a computing device in that it specifies a valuation transformer (i.e., a function $\overline{\mathcal{A}} : 2^{\mathcal{V}} \rightarrow 2^{\mathcal{V}}$) which given valuation V returns a new valuation

$\overline{\mathcal{A}}(V)$ that contains those variables which are set by a clause in $\mathcal{A}$ (under V). The transition from V to $\overline{\mathcal{A}}(V)$ will represent one reaction (instant of time) in computations of $\mathcal{A}$.

$\mathcal{A}$ is equipped with an additional infrastructure in that set $\mathcal{V}$ is partitioned into sets $\mathcal{V}_S$ and $\mathcal{V}_C$ of *signal* and *control* variables, respectively. Similarly, clauses are partitioned into the sets of *signal* and *control* clauses, depending on what kind of variable they define. The following notation is used for the different kinds of clauses:

$$\begin{array}{ll} s \Leftarrow b & \text{(signal clause)} \\ c \leftarrow b & \text{(control clause)} \end{array}$$

Signal variables correspond to signals manipulated by the program and a signal clause $s \Leftarrow b$ states that s is emitted if b is true. Signal variables which are not defined correspond to inputs of the program. Signal s can be tested for presence/absence by using s on the left-hand side of a clause. Valuations of signal variables represent so-called *events*. An event $E \in 2^{\mathcal{V}_S}$ contains those signals which are present during a reaction.

Control variables correspond to control points in the program and a control clause $c \leftarrow b$ states that a specific control point is reached when b is true. The current (possibly distributed) state of the program is specified by a set of currently reached control points, thus by a valuation of control variables $\sigma \in 2^{\mathcal{V}_C}$.

Formally, $\mathcal{A}$ specifies a labeled transition system $(\Sigma, \emptyset, \Lambda, \rightarrow)$ with the set of states $\Sigma = 2^{\mathcal{V}_C}$, the set of labels $\Lambda = 2^{\mathcal{V}_S} \times 2^{\mathcal{V}_S}$, and the transition relation $\rightarrow \subseteq \Sigma \times \Lambda \times \Sigma$ defined by the following rule:

$$\sigma \xrightarrow{E/E'} \sigma' \quad \text{iff} \quad E' \subseteq E \text{ and } \overline{\mathcal{A}}(\sigma \cup E) = \sigma' \cup E'$$

In the initial state of $\mathcal{A}$, all control variables are false.

4.1 Operations on Boolean automata

Let $\mathcal{P}.v$, for a Boolean automaton $\mathcal{P}$ and variable v, denotes the term defining v. (Formally, if $v := b \in \mathcal{P}$ then $\mathcal{P}.v \equiv b$, otherwise $\mathcal{P}.v \equiv \mathit{ff}$.) Using this notation we can define the following operations on Boolean automata:

$$\begin{array}{ll} \text{(disjunction)} & (\mathcal{P} \vee \mathcal{Q}).v = (\mathcal{P}.v) \vee (\mathcal{Q}.v) \\ \text{(conjunction)} & (b \wedge \mathcal{P}).v = b \wedge (\mathcal{P}.v) \\ \text{(substitute)} & (\mathcal{P}[b/x]).v = (\mathcal{P}.v)[b/x] \\ \text{(replace)} & (\mathcal{P} \triangleleft x := b).v = \begin{cases} b & \text{if } v \equiv x \\ \mathcal{P}.v & \text{otherwise} \end{cases} \end{array}$$

In addition, let $\mathcal{P}[x//y]$, for variables x and y, denote renaming of y by x in the whole automaton $\mathcal{P}$ (both on left-hand and right-hand sides of all clauses in $\mathcal{P}$).

In the sequel, whenever we write $\mathcal{P} \vee \mathcal{Q}$ we always assume that $\mathcal{P}$ and $\mathcal{Q}$ have disjoint state spaces, which can easily be accomplished by a proper renaming of control variables, in either $\mathcal{P}$ or $\mathcal{Q}$. Notice that under such an assumption, $\mathcal{P} \vee \mathcal{Q}$ describes synchronous parallel composition $\mathcal{P}||\mathcal{Q}$.

Finally, let $\overset{\Leftarrow}{\mathcal{P}}$ and $\overleftarrow{\mathcal{P}}$ denote all signal and control clauses in $\mathcal{P}$, respectively.

4.2 $\alpha\eta$-split

We assume that every Boolean automaton $\mathcal{P}$ has a special input variable α, called a start variable. We further assume that α is present only in the first instant (reaction, transition) of any computation of $\mathcal{P}$. In this way, we can easily distinguish the first reaction from all later reactions, in all computations of $\mathcal{P}$. In the first reaction, all control variables are false. In all later reactions α is false. In the sequel, we use the following derived operations:

$$\begin{array}{ll} (\alpha\text{-part}) & \mathcal{P}_\alpha = \mathcal{P}[\mathit{ff}/\mathcal{V}_C] \\ (\eta\text{-part}) & \mathcal{P}_\eta = \mathcal{P}[\mathit{ff}/\alpha] \end{array}$$

$\mathcal{P}_\alpha$ specifies behaviour of $\mathcal{P}$ in the first instant while $\mathcal{P}_\eta$ specifies behaviour of $\mathcal{P}$ in all later instants.

5 Translating LEA to Boolean automata

A program in LEA consists of a set of declarations. Each declaration specifies one reactive component which we call a compilation unit (or just unit, in short). In original Lustre, Esterel and Argos, units are called nodes, modules and processes, respectively. A unit has an interface which specifies its name, inputs and outputs, and a body which specifies its reactive behaviour. Units are declared by the following phrase:

$$\mathsf{unit}\ U\ (\mathit{inputs})\ (\mathit{outputs});\ \mathit{body}$$

In Argos and Esterel, *body* is a statement while in Lustre it is a set of equations. (Argos statements have both textual and visual representations. In this paper, we use the textual representation only.)

Unit U is translated to a Boolean automaton denoted by $\overline{U}$. Every $\overline{U}$ defines two special output variables ω and η. Variable ω is meant to be true in the instant U terminates. Variable η represents the disjunction of all control variables in $\overline{U}$. Since in the first instant of executing $\overline{U}$, no control variable is true, η is true only in later instants of executing $\overline{U}$.

We present the translation rules in two steps. In the first step, we consider the three LEA languages in separation, and describe how to translate *body* depending on the language it is written in. In particular, we will not cover import statements, in this first step. In the second step, we present uniform translation rules for the import statements.

Remark In our presentation of LUSTRE, ESTEREL and ARGOS, we only concentrate on essential concepts employed in the languages. In particular, we do not pay much attention to be in agreement with the original syntax of the languages.

6 Translating ESTEREL to Boolean automata

In our version of ESTEREL, traps are treated in a more general way than in original ESTEREL. They are simply local signals of a special kind. An ESTEREL statement has one of the following forms:

$$\begin{array}{l}
\mathsf{emit}\ s \\
\mathsf{exit}\ s \\
\mathsf{skip} \\
\mathsf{wait} \\
P\,;Q \\
\mathsf{loop}\ P\ \mathsf{end} \\
\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{end} \\
P\,||\,Q \\
\mathsf{abort}\ P\ \mathsf{when}\ b \\
\mathsf{trap}\ s\ \mathsf{in}\ P\ \mathsf{end} \\
\mathsf{local}\ s\ \mathsf{in}\ P\ \mathsf{end}
\end{array}$$

where P and Q are statements, s is a signal's name, and b is a Boolean term over signal names.

A statement P is started in some instant, called the first instant of executing P, and then it may either terminate in the first instant or continue its execution in later instants. In the former case, the statement is called *instantaneous*. During its execution a statement may emit signals, either unconditionally or in response to some signals being present or absent, in the instant.

$\mathsf{emit}\ s$ and $\mathsf{exit}\ s$ emit signal s in the same instant they are started. $\mathsf{emit}\ s$ is instantaneous while $\mathsf{exit}\ s$ never terminates.

skip and wait do not emit signals. skip is instantaneous while wait never terminates.

$P;Q$ denotes instantaneous sequential composition. Q is started in the same instant in which P terminates.

$\mathsf{loop}\ P\ \mathsf{end}$ is an infinite iteration of P. It is equivalent to $P;\mathsf{loop}\ P$.

$\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{end}$ behaves either as P or as Q, depending on whether guard b is true or false, in the first instant of executing the if statement. b is true if it evaluates to tt under the valuation which interprets signal s as true iff s is present.

emit s	$\{\sigma \Leftarrow \alpha,\ \omega \Leftarrow \alpha\}$
exit s	$\{\sigma \Leftarrow \alpha,\ \omega \Leftarrow ff\}$
skip	$\{\omega \Leftarrow \alpha\}$
wait	$\{\ c \leftarrow \alpha \vee c,\ \eta \Leftarrow c\}$
$P\,;Q$	$\mathcal{P} \vee (\mathcal{Q}[\mathcal{P}.w/\alpha])$
loop P	$\mathcal{P}[\alpha \vee \mathcal{P}.\omega/\alpha] \triangleleft \omega \Leftarrow ff$
if b then P else Q	$\mathcal{P}[\alpha \wedge b/\alpha] \vee \mathcal{Q}[\alpha \wedge \neg b/\alpha]$
$P \,\|\|\, Q$	$(\mathcal{P} \vee \mathcal{Q}) \triangleleft \omega \Leftarrow (\mathcal{P}.\omega \wedge \mathcal{Q}.\omega) \vee (\mathcal{P}.\omega \wedge \neg \mathcal{Q}.\eta) \vee (\mathcal{Q}.\omega \wedge \neg \mathcal{P}.\eta)$
abort P when b	$(\mathcal{P}_\alpha \vee \neg b \wedge \mathcal{P}_\eta) \vee \{\omega \Leftarrow b \wedge \mathcal{P}.\eta\}$
trap s in P end	$\overset{\Leftarrow}{\mathcal{P}} \vee (\neg \tau \wedge \overleftarrow{\mathcal{P}}) \vee \{\omega \Leftarrow \tau\}$
local s in P end	$\mathcal{P}$

Table 1: Translation of Esterel.

$P \,||\, Q$ denotes synchronous parallel composition. P and Q are both started in the first instant of executing $P \,||\, Q$ and then continue their own executions. $P \,||\, Q$ terminates in the first instant in which both P and Q are terminated.

Let $Q \equiv$ abort P when b. Q is called a strong pre-emption statement. In the first instant of its execution, Q behaves as P, and if P terminates so does Q. If P does not terminate, in the first instant, its execution is continued till either P terminates or its execution reaches the instant in which b is true. In both cases Q terminates. However, in the later case, P is immediately interrupted without emitting anything it could otherwise emit.

Let $Q \equiv$ trap s in P end. Q is called a weak pre-emption statement. It behaves as P till either P terminates, in which case Q terminates as well, or some exit s statement is reached during execution of P. In such a case, s is emitted and P is weakly pre-empted causing Q to terminate, but only if no other signal, which is trapped in an outer trap statement, is emitted from P.

6.1 Translation rules

The translation rules are defined inductively in table 1. Translation of statements P and Q is denoted by $\mathcal{P}$ and $\mathcal{Q}$, respectively. The rules are proved to be correct, in [6], with respect to the formal semantics of pure Esterel [1].

Each statement P, or rather its translation $\mathcal{P}$, defines two special output variables η and ω. (Notice that not all rules in table 1 define the variables, but recall that we use the convention that if a variable v is not defined then its definition is $v := \mathit{ff}$.)

Variable η represents the disjunction of all control variables in $\mathcal{P}$. Notice that only wait statements introduce control variables, in the clause $c \leftarrow \alpha \vee c$. Thus, η is true in an instant if P was waiting in the previous instant.

Variable ω is (guaranteed to be) true in the instant the statement terminates. The terminate condition is needed in the rules for $P\,;\,Q$ and loop P end. In $P\,;\,Q$, $\mathcal{P}.w$ is substituted for α, in $\mathcal{Q}$, to start Q when P terminates. In loop P end, $\mathcal{P}.w$ is used to restart P after every iteration of the loop.

In $P \,||\, Q$, ω is redefined to capture the following termination condition: either P and Q terminate simultaneously, or one of them terminates later than the other.

In abort P when b, P should be aborted if b is true, but only in later instants. That is why $\overline{P}$ is split into $\alpha\eta$-part and only the η-part is guarded by $\neg b$. The guard ensures that no signals are emitted and all control variables are deactivated. In the instant in which P is aborted, the whole abort statement should terminate and that is why the clause $\omega \Leftarrow b \wedge \mathcal{P}.\eta$ is added.

For example, the statement abort wait when s (await s, in full Esterel) translates (after several Boolean simplifications) to

$$
\begin{aligned}
&c \leftarrow \alpha \vee (\neg s \wedge c)\\
&\eta \Leftarrow c\\
&\omega \Leftarrow s \wedge c
\end{aligned}
$$

In trap s in P end, τ denotes the condition which is true when s is present (i.e., some exit s statement is reached during execution of P), and no outer trap signals are emitted inside P. In such a case, the trap statement should terminate (see $\omega \Leftarrow \tau$) and P should be weekly pre-empted (see $\neg\tau \wedge \overleftarrow{\mathcal{P}}$ which deactivates all control variables in $\mathcal{P}$). If $s_1, \ldots, s_k$ denote all outer trap signals such that exit s_i appears inside P then τ stands for the term $\mathcal{P}.s \wedge \neg(\mathcal{P}.s_1 \vee \cdots \vee \mathcal{P}.s_k)$.

6.2 Reincarnation problem

In fact, the translation rule for the loop statement is more complicated than the one given in table 1, due to the phenomenon known as reincarnation of local signals. Compare the following two statements:

```
local s in                              loop
  loop                                    local s in
    if s then emit o else skip end;         if s then emit o else skip end;
    await true;                             await true;
    emit s;                                 emit s;
  end;                                    end;
end;                                    end;
```

local s in ... introduces a fresh copy of s, every time it is entered, and await $true$ delays execution by one instant.

In the first statement, signal o is present in all instants, except the first one. In the second statement, signal o is always absent since the emit s statement does not influence the if s ... statement.

Our translation rules does not capture the difference. Both statements translate (after several Boolean simplifications) to

$$
\begin{aligned}
&o \Leftarrow s \wedge (\alpha \vee c)\\
&c \leftarrow \alpha \vee c\\
&s \Leftarrow c\\
&\eta \Leftarrow c\\
&\omega \Leftarrow f\!f
\end{aligned}
$$

which is correct for the first example only.

To remedy the problem, all local signals inside loop P end have to be reincarnated, in the sense that each local signal must be represented by two variables. A local signal, say s, is represented by variables s^α and s^η. Variable s^α represents the status of s in the first instant of executing P while s^η represents the status of s in all later instants.

Assuming s is the only local signal in loop P end, the loop is translated to

$$(\mathcal{P}_\alpha[s^\alpha//s] \vee \mathcal{P}_\eta[s^\eta//s])[a \vee \mathcal{P}.\omega/\alpha]$$

Using the new rule, the second example translates correctly to

$$
\begin{aligned}
&o \Leftarrow f\!f\\
&c \leftarrow \alpha \vee c\\
&s^\alpha \Leftarrow f\!f\\
&s^\eta \Leftarrow c\\
&\eta \Leftarrow c\\
&\omega \Leftarrow f\!f
\end{aligned}
$$

7 Translating Argos to Boolean automata

An Argos statement has one of the following forms:

$$\mathsf{skip},\ P\,||\,Q,\ \mathsf{local}\ s\ \mathsf{in}\ P\ \mathsf{end},\ \mathsf{auto}\ \mathit{states}\ \mathit{trans}\ \mathsf{end}$$

Argos statements never terminate. skip, || and local are as in Esterel (modulo termination). The auto statement specifies a reactive automaton.

The reactive automaton consists of states and transitions. States are control points: control can enter and leave a state. Each transition is a triple (*source*, *target*, *label*). It connects two states, *source* and *target*, and is labeled by a pair *guard*/*signal* where *guard* is a Boolean term over signal names. A transition is taken if control is in its source state and its *guard* is true. When transition is taken the *signal* is emitted, control leaves the source state and enters the target state. Such a transition takes exactly one instant.

One of the states is distinguished as an initial state. When the automaton is started it performs an implicit transition to the initial state.

Argos automata are hierarchic: a state can be refined by a statement. In fact, we assume that all states are refined, since the skip statement can be used for a "dummy" refinement. When the refined state is entered the statement is started. When it is left the statement is (weakly) pre-empted.

The only translation rules specific to Argos are concerned with translation of Argos automata. Let $\mathcal{A}$ denote the following Argos automaton:

state $s_1 = P_1$
state $s_2 = P_2$
...
tran t_1 from p_1 to q_1 with g_1/o_1
tran t_2 from p_2 to q_2 with g_2/o_2
...

where P_i is the statement by which state s_i is refined.

With each state s we associate a control variable, say s, which is true when control is in the state. Similarly, with each transition t we associate a (local) signal variable, say t, which is present when the transition is taken.

Each transition tran t from p to q with g/o is translated to

$$\overline{t} = \{t \Leftarrow p \wedge g,\ o \Leftarrow t\}$$

Each state state $s = P$ is translated to

$$\overline{s} = \{s \leftarrow in_s \vee (\neg out_s \wedge s)\}$$

where $in_s \equiv \bigvee t_j$, for all transitions t_j which enter s, and $out_s \equiv \bigvee t_k$, for all transitions t_k which leave s. (If s is initial then $in_s \equiv \alpha \vee \bigvee t_j$.)

Refinement of state s by statement P is translated as

$$\overline{r} = (\overset{\Leftarrow}{\mathcal{P}} \vee \overset{\leftarrow}{\mathcal{P}}_\alpha \vee (\neg out_s \wedge \overset{\leftarrow}{\mathcal{P}}_\eta))[in_s/\alpha]$$

Finally, the automaton $\mathcal{A}$ is translated to

$$\overline{\mathcal{A}} = (\bigvee \overline{t_i}) \vee (\bigvee \overline{s_j}) \vee (\bigvee \overline{r_j}) \lhd \eta \Leftarrow \bigvee s_i$$

8 Translating Lustre to Boolean automata

We consider only pure Lustre without so-called clocks. In Lustre, a behaviour of a reactive component is specified by a set of equations $s = e$, for each output or local signal s. e is an expression built from signal names (inputs included) and various Lustre operators. The expression evaluates to tt or ff, and the equation $s = e$ has the following meaning: in every instant of time, signal s should be emitted if e is true.

We restrict e, without loss of expressiveness in comparison with the original Lustre, to have one of the following forms: b, b_1 -> b_2 and pre b, where b, b_1 and b_2 are Boolean terms. In the first instant, the value of b_1 -> b_2 is the value of b_1, and in all later instants it is the value of b_2. In the first instant, pre b is false while in all later instants it has the value of b from the previous instant.

Lustre expressions can have yet another form, of so-called *node applications*, but we postpone their discussion to section 9.

For example, the following declaration specifies a unit which recognizes a rising edge in a supplied input signal:

```
unit redge (signal) (edge);
  local CurrentlyUp, PreviouslyDown in
    edge = signal -> (CurrentlyUp ∧ PreviouslyDown);
    CurrentlyUp = signal;
    PreviouslyDown = pre (¬ signal);
  end
```

Equations can be mutually recursive, but recursion must be guarded by pre (otherwise the equations either could not be "solved" or would have many "solutions").

With each Lustre unit we associate a control variable, say c. A Lustre *body*, say equations $eq_1; \ldots ; eq_n$, is translated to

$$(\overline{eq}_1 \vee \cdots \vee \overline{eq}_n) \lhd \{c \leftarrow \alpha \vee c,\ \eta \Leftarrow c\}$$

where $\overline{eq}$ is the result of translating eq, as depicted in table 2. The control variable m (memory), defined in the translation rule for $x =$ pre b, is treated in a special way. It is not regarded as a control variable when $\alpha\eta$-split is performed.

$x = b$	$\{x \Leftarrow (\alpha \vee c) \wedge b\}$
$x = b_1$ -> b_2	$\{x \Leftarrow \alpha \wedge b_1 \vee c \wedge b_2\}$
$x = \mathsf{pre}\ b$	$\{x \Leftarrow (\alpha \vee c) \wedge m,\ m \leftarrow (\alpha \vee c) \wedge b\}$

Table 2: Translation of LUSTRE equations

9 Integration of LUSTRE, ESTEREL and ARGOS

In ESTEREL, we use the following statement to import unit U with inputs $i_1, \ldots, i_m$ and outputs $o_1, \ldots, o_n$:

$$\mathsf{run}\ U\ (b_1, \ldots, b_m)\ (s_1, \ldots, s_n)$$

As previously, b_j are Boolean terms and s_k are signal names. The intended meaning of the run statement is to make input i_j present whenever b_j is true and to emit s_k whenever U emits output o_k.

Notice that our run statement differs from an analogous statement in original ESTEREL. We have generalized it to allow passing *expressions* in place of formal input parameters, and not only *signal names*. In other words, our input parameters are passed "by value" instead of "by name".

Since original ARGOS lacks such a general concept of import, we have simply extended the repertoire of ARGOS statements by the run statement.

In LUSTRE, one can use the following equation to import U:

$$(s_1, \ldots, s_n) = U\ (b_1, \ldots, b_m)$$

A simple approach would be to translate run into

$$\overline{U}[b_1/i_1, \ldots, b_m/i_m][s_1//o_1, \ldots, s_n//o_n]$$

with suitable renaming of local variables in $\overline{U}$, to avoid name clashes. Since the rule captures quite well the original semantics of import in ESTEREL and LUSTRE, it works fine if there is no cross-language import. Unfortunately, the rule gets incorrect when a LUSTRE unit, say L, is imported into ESTEREL (or ARGOS), since the context into which L is imported may influence the behaviour of L via L's outputs, and this is what the semantics of LUSTRE forbids. For an example of such situation, let us consider the following LUSTRE unit

$$\mathsf{unit}\ L\ ()\ (x);\ x = \mathit{ff}\ \text{->}\ (\mathsf{pre}\ x)$$

Obviously, the intended meaning is that x should always be false, but if L is imported into ESTEREL, as in emit x || run $L()(x)$, then x becomes always true. (Notice that this example cannot be replayed in LUSTRE alone.)

The following translation rule

$$\overline{U}[b_1/i_1, \ldots, b_m/i_m] \triangleleft \{s_1 \Leftarrow o_1, \ldots, s_n \Leftarrow o_n\}$$

makes outputs of U flow in one direction only, and this is enough to remedy the problem.

10 Conclusions

We have presented a framework in which LUSTRE, ESTEREL and ARGOS can be compiled in a uniform way. As a consequence, one gets a smooth integration of the languages, and this allows to combine the three languages into one multi-paradigm language which we call LEA.

Although we have considered, in this paper, only a simplified version of LEA, just to illustrate the main ideas behind our approach to integrating LUSTRE, ESTEREL and ARGOS, the approach can easily be extended to cover full LEA. In particular, our translation rules can be used almost without change to handle valued signals which carry data of more complicated types like integer and real numbers. It turns out that in order to handle values it is enough to generalise pure signal clauses $s \Leftarrow b$ to valued signal clauses $s \Leftarrow e$ **if** b, where e is an expression whose value is emitted on signal s when b is true. The only difficult step in going from pure LEA to valued LEA is to correctly handle the reincarnation problem.

Full LEA has been (almost fully) implemented in Synchronie Workbench. The translation rules used in SWB are more complicated than the ones presented here since we had to take into account the efficiency of a final generated code. For example, the costly operation of $\alpha\eta$-split can be removed, using various tricks, from most of our translation rules. It needs to remain only in the rule for loop statement, to solve the reincarnation problem.

This technical summary lacks any comparison of our work with some other attempts to integrating synchronous languages. We will include such a comparison in the full paper. In short, other attempts have concentrated mainly on sharing tools, via using the same intermediate format, while our approach allows to share the languages themselves.

References

[1] G. Berry. A hardware implementation of pure Esterel. *Sadhana, Academy Proceedings in Engineering Sciencies, Indian Academy of Sciences*, 17:95–130, 1992.

[2] G. Berry and G. Gonthier. The synchronous programming language Esterel: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[3] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Dordrecht, 1993.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1321, Sept. 1991.

[5] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proceedings of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, Springer-Verlag, 550–564, Aug. 1992.

[6] A. Poigné and L. Holenderski. Boolean automata for implementing pure Esterel. Arbeitspapiere der GMD 964, Forschungszentrum Informationstechnik GmbH, 1–47, Dec. 1995.

[7] Synchronie Workbench, `http://set.gmd.de/~lhol/swb`, May 1997.