# Compiling Argos into Boolean equations

(Extended Abstract)

F. Maraninchi, N. Halbwachs

VERIMAG,* Miniparc – ZIRST, F-38330 Montbonnot
email: Florence.Maraninchi@imag.fr
tel: (33) 76.90.96.42, fax: (33) 76.41.36.20

**Abstract**

In most imperative synchronous languages (Esterel, Argos, Statecharts,...), the semantics of the control structures may be conveniently described as compositions of Mealy machines. This constitutes the usual formal semantics of Argos, for instance, where basic components are Mealy machines. On the other hand, the compilation process should not be based upon an explicit generation of the Mealy machine that represents the behaviour of the whole program, because this machine may have a very large number of states. Hence we try to perform a symbolic compilation into a Mealy machine implicitly represented by a set of Boolean equations.

We give here the direct semantics of Argos in terms of such equations, and show that this semantics coincides with the usual one. The current implementation of the Argos compiler produces DC code, which is the common equational format for synchronous languages. This will allow to merge imperative and declarative synchronous languages (Argos and Lustre, for instance), by merging DC files.

*Verimag is a joint laboratory of CNRS, Institut National Polytechnique de Grenoble, Université J. Fourier and Verilog SA associated with IMAG.

# 1 Introduction

Imperative synchronous languages like Esterel [BG92, IEE91], Argos [Mar92, Hal93], Statecharts [Har87] and others, provide sets of control structures whose semantics can be conveniently described using Mealy machines and operations on such machines, like the Cartesian product [JM94].

For ARGOS or Statecharts, where explicit descriptions of Mealy machines constitute the basic components of programs, such a semantics is operational: the behaviour of a complex program is a single Mealy machine, that can be obtained by composing the machines associated to the components. However, if applied without care, this idea yields a very inefficient compiling technique. Indeed, the Mealy machine that represents the behaviour of the whole program may have a very large number of states, and its explicit generation may have a prohibitive cost. The size of the global machine is intrinsically exponential in the size of the source program.

On the other hand, the DC code [CS95], which has been defined as a common format for all synchronous languages, provides an equational description of reactive systems, in the style of LUSTRE. While LUSTRE has *clocks*, DC has a simpler notion of *activation condition*: executing a program can be done by a cyclic evaluation of the equations, as in a synchronous circuit, but the program may specify that some equations do not need to be evaluated at each step. This is particularly interesting when trying to translate ARGOS into DC, since the hierarchy operator of ARGOS is a way of specifying that some parts of the program are sometimes irrelevant for the global behaviour.

In this paper, we propose to define the imperative synchronous constructs of ARGOS in terms of Boolean equations. G. Berry has already defined the semantics of pure ESTEREL in terms of Boolean equations [Ber91, Ber]. It provides an efficient compilation technique, and clarifies the notion of causality. For ARGOS it will be much simpler, as far as the language constructs are concerned. The interesting point is the way we use activation conditions. We give here the direct translation of Argos into DC, and show that this semantics coincides with the usual one.

Translating LUSTRE into DC is straightforward. It is under development in the compiler. When both ARGOS and LUSTRE programs are translated into DC, it will be possible to merge them, by merging DC programs. The node structure of DC ensures the appropriate amount of encapsulation for that to make sense [JLMR94].

The paper is organized as follows: section 2 defines Boolean Mealy machines and the set of operations used in the formal semantics of ARGOS; section 3 presents the Boolean subset of the DC format we need for encoding ARGOS behaviour expressions into equations; section 4 describes the translation, for correct programs; section 5 is a short example. section 6 sketches the proof that the semantics in terms of Mealy machines operations, and the semantics in terms of Boolean equations, coincide; Finally section 7 gives some ideas about implementation and further work.

# 2 Imperative Synchronous Constructs as Compositions of Boolean Mealy Machines

ARGOS [Mar92] can be viewed as a set of operators on Mealy machines. This is the easiest and the most primitive way to define imperative synchronous constructs, which can be given a natural graphical syntax inspired from Statecharts. The semantics is perfectly synchronous. We first give the set of operations on Mealy machines that constitute the semantics of the ARGOS operators, when programs are correct. Then we define the semantics of ARGOS programs in terms of these operations, introducing the notion of *incorrect* programs.

## 2.1 Boolean Mealy Machines

We consider a global alphabet of *signal* names $\mathcal{A}$. A Mealy machine is a tuple $(S, s_0, I, O, T)$ where $I \subseteq \mathcal{A}, O \subseteq \mathcal{A}$ are the sets of *input* and *output* signals; $S$ is the set of states; $s_0$ is the initial state; $T \subseteq S \times \mathcal{B}(I) \times 2^O \times S$ is the set of transitions ($\mathcal{B}(I)$ denotes the set of Boolean formulas with variables in $I$. Without loss of generality, we can consider that these formulas are *monomials*, i.e. conjunctions of signals or negations of signals). The set of Mealy machines is denoted by $\mathcal{M}$.

Simple systems are described directly by Mealy machines (see Figure 1). State names have no semantics; the initial state is distinguished by a little arrow; transition labels are input/output-structured: the input part is a Boolean formula whose variables are *input signals*, the output is a set of *output signals*. The system of Figure 1 emits the signal $X$ every two $A$'s, provided $B$ is present at the same instant.

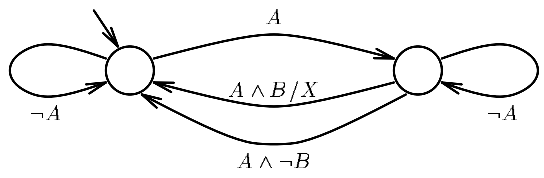


Figure 1: A simple system described by a Mealy machine

## 2.2 Operations on Mealy machines

### 2.2.1 Synchronous Product

Figure 2 shows a simple parallel program with no synchronization between the components: the system described emits $A$ every two $a$'s and $B$ every two $b$'s. The parallel composition is associative and commutative. It combines the components of figure 2 into the single machine of figure 3.

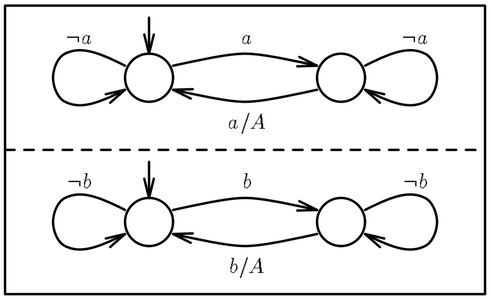


Figure 2: Parallel composition with no synchronization

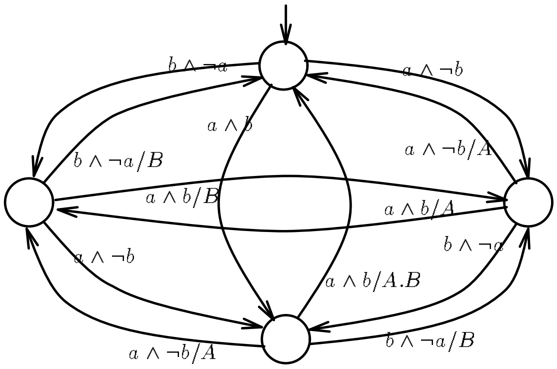


Figure 3: Expanding the parallel composition

This composition is formally described by the operation $\times \quad : \quad \mathcal{M} \times \mathcal{M} \longrightarrow \mathcal{M}$.
$(S_1, s_{01}, I_1, O_1, T_1) \times (S_2, s_{02}, I_2, O_2, T_2) = (S_1 \times S_2, (s_{01}, s_{02}), I_1 \cup I_2, O_1 \cup O_2, T')$
Where $T'$ is defined by:
$((s_1, m_1, o_1, s_1') \in T_1, (s_2, m_2, o_2, s_2') \in T_2) \iff ((s_1, s_2), m_1 \wedge m_2, o_1 \cup o_2, (s_1', s_2')) \in T')$

### 2.2.2 Encapsulation

Two components X and Y may be synchronized if an input $\ell$ of X is an output of Y. To express that the signal $\ell$ should be used only for synchronizing $X$ and $Y$, one declares it to be *local* to the parallel composition of $X$ and $Y$. Figure 4 shows a system which emits $A$ every four $a$'s (a two-bit counter). $\ell$ is local to the parallel composition of two one-bit counters. The resulting machine is given by Figure 5.

The encapsulation is described by the operation $\setminus$ : $\quad \mathcal{M} \times 2^{\mathcal{A}} \longrightarrow \mathcal{M}$

$(S, s_0, I, O, T) \setminus \Gamma = (S, s_0, I \setminus \Gamma, O \setminus \Gamma, T')$ where $T'$ is defined by:
$(s, m, o, s') \in T \ \wedge \ m^+ \cap \Gamma \subseteq o \ \wedge \ m^- \cap \Gamma \cap o = \emptyset \quad \Longleftrightarrow \ (s, \exists\Gamma.m, o \setminus \Gamma, s') \in T'$

$m^+$ denotes the set of variables that appear as positive in the monomial $m$ (i.e. $m^+ = \{x \in \mathcal{A} \mid (x \wedge m) = m\}$). $m^-$ denotes the set of variables that appear as negative in the monomial $m$ (i.e. $m^- = \{x \in \mathcal{A} \mid (x \wedge m) = \texttt{false}\}$).

Intuitively, a transition $(s, m, o, s') \in T$ remains in the encapsulated program, if and only if its label passed the two following criteria: $m^+ \cap \Gamma \subseteq o$, which means that the signals of $\Gamma$ that are supposed to be present, are indeed emitted; and $m^- \cap \Gamma \cap o = \emptyset$, which means that the signals of $\Gamma$ that are supposed not to be present, should not be emitted. If the transition passes these two criteria, its label is modified: elements of $\Gamma$ are hidden, both in the input condition and in the output set.

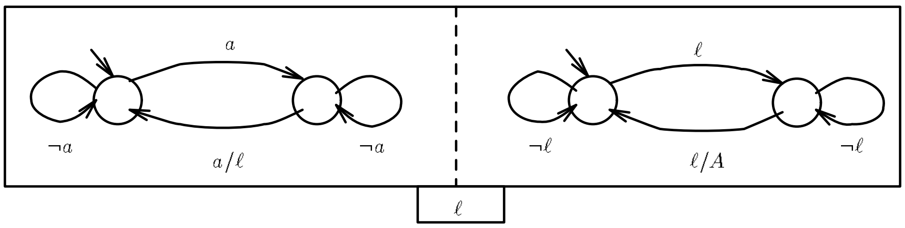


Figure 4: Synchronization and local signals

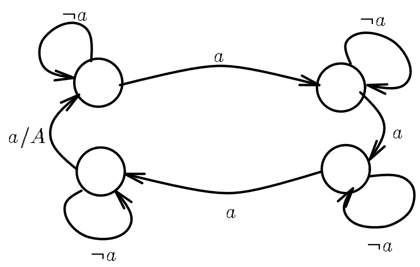


Figure 5: Expanding the local signal declaration

### 2.2.3 Refinement

Figure 6 shows how to add *start* and *stop* functionalities to the modulo 4 $a$-counter described before. The resulting machine is given by Figure 7. A state of a machine can be *refined* by any composed process. When a refined state is entered, the process is started in its initial state; when it is left, the process is killed. Hence each machine can be viewed as a kind of *controller* which may start and kill subprocesses.

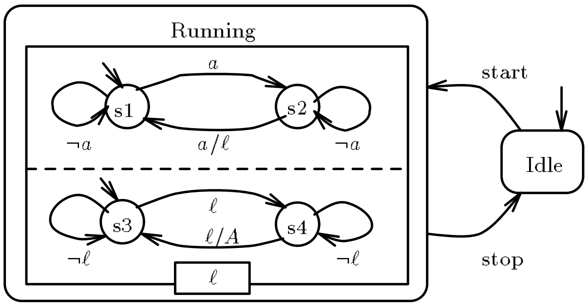


Figure 6: Refinement

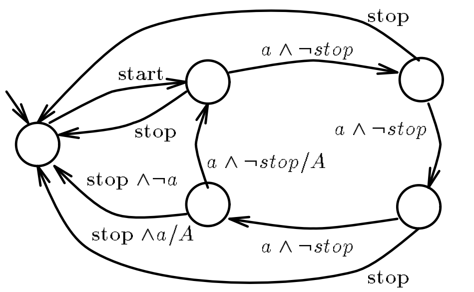


Figure 7: Expanding the refinement

This composition is described by the operation $\triangleright$, applied to a machine used as controller, and a set of refining machines:

$\triangleright \quad : \quad \mathcal{M} \times 2^{\mathcal{M}} \longrightarrow \mathcal{M}$

Let us note $M = (S, s_0, I, O, T)$ where $S = \{s_0, s_1, ..., s_n\}$. And consider a set of refining machines $\{M_j\}_{j=0..n}$, where $M_j = (S^j, s_0^j, I^j, O^j, T^j)$, and $S^j = \{s_0^j, s_1^j, ..., s_{n_j}^j\}$.

The composed machine, in which $M_j$ refines $s_j$, is: $M \triangleright \{M_j\}_{j=0..n} = (S \triangleright \{S^j\}_J, s_0 \triangleright s_0^0, I \cup \bigcup I^j, O \cup \bigcup O^j, T')$. Its set of states is of the form:

$S \triangleright \{S^j\}_J = \bigcup_{j=0}^{n} \{s_j \triangleright s_k^j, k \in [0..n_j]\}$
and its transitions $T'$ are obtained by the two following rules:

1) a transition from $s_a$ to $s_b$ in the controller, together with a transition from $s_k^a$ to $s_{k'}^a$ in the currently active refining machine: the outputs are added; in the global target state, the machine that refines $s_b$ is started in its initial state $s_0^b$. The machine that refines $s_a$ has been killed. Its state is irrelevant.

$(s_a, m, o, s_b) \in T \quad \wedge \quad (s_k^a, m', o', s_{k'}^a) \in T^a \qquad \Longrightarrow \quad (s_a \triangleright s_k^a, m \wedge m', o \cup o', s_b \triangleright s_0^b) \in T'$

2) a transition in the currently active refining machine, from $s_k^a$ to $s_{k'}^a$, if no transition sourced in $s_a$ is enabled in the controller.

$$(s_k^a, m', o', s_{k'}^a) \in T^a \quad \Longrightarrow \quad (s_a \triangleright s_k^a, m' \wedge \left[ \bigwedge_{(s_a, m, -, -) \in T} \neg m \right], o', s_a \triangleright s_{k'}^a) \in T'$$

### 2.3 Behaviour expressions and semantics in terms of Mealy machines

The set $\mathcal{E}$ of behaviour expressions we consider is defined by:

$$E \quad ::= \quad E \| E \mid \overline{E^{\Gamma}} \mid \mathbf{R}_M(R_1, ..., R_n) \qquad R ::= E \mid \mathsf{NIL}$$

$\mathsf{NIL}$ is introduced to express that a state is not refined. Semantically, it's a one-state process which has no inputs and emits nothing.

Not all combinations of machines are considered as correct expressions. All basic components must be deterministic. We also reject expressions containing a sub-expression whose associated machine is non deterministic. The detection of incorrect expressions is not considered in this paper; see [Mar92, HM95, Ber].

The semantic function: $\mathcal{S} : \mathcal{E} \longrightarrow \mathcal{M}$ is defined recursively by:

$$\mathcal{S}(E_1 \| E_2) = \mathcal{S}(E_1) \times \mathcal{S}(E_2)$$
$$\mathcal{S}(\mathbf{R}_M(R_1, ..., R_n)) = M \triangleright (\mathcal{S}(R_1), ...\mathcal{S}(R_n))$$
$$\mathcal{S}(\overline{E^{\Gamma}}) = \mathcal{S}(E) \setminus \Gamma$$
$$\mathcal{S}(\mathsf{NIL}) = (\{\mathsf{NIL}\}, \mathsf{NIL}, \emptyset, \emptyset, \{(\mathsf{NIL}, \mathsf{true}, \emptyset, \mathsf{NIL})\})$$

# 3 Structured Boolean Equations

We use a Boolean subset of the DC format [CS95]. A DC program describes a reactive machine. Informally, the behaviour of such a machine along the time is an infinite sequence of reactions. This sequence of reactions defines the sequence of *instants* that the program perceives. The main objects handled by a DC program are *flows*. A flow is a typed object which holds a value at each instant of the program life. If x is a flow, we will note $x_n$ its value at the $n$th reaction (or $n$th *instant*) of the program.

A program receives input flows and computes output flows, possibly using local flows which are not visible from the environment. In DC, a constant is considered as a constant flow, always holding the same value. Local and output flows can be defined by *definitions*.

More precisely, a flow definition is given an *activation condition*, which is a Boolean flow. The meaning is that the defined flow can be updated only when the activation condition has the value *true*, otherwise it keeps its preceding value.

We consider two kinds of definitions: equations (EQU's in the sequel) and memorizations (MEMO's).

EQU$(i, b, a)$ defines a flow $i$ to be the result of the Boolean expression $b$, updated when the expression $a$ is true:

$$i_0 = \begin{cases} b_0 \text{ if } a_0 \\ v \text{ otherwise} \end{cases}, \qquad i_n = \begin{cases} b_n \text{ if } a_n \\ i_{n-1} \text{ otherwise} \end{cases} \text{ when } n > 0$$

We should add a parameter to the definition of an EQU, for the initial value $v$. However, this value serves only if the activation condition $a$ is false at the first instant. We shall see that, in the translation of ARGOS, a flow defined by an EQU, whose activation condition is false at the first instant, is not used anywhere. Hence the initial values of EQU's are always irrelevant.

MEM$(i, b, a, v)$ defines a flow $i$ to be the memorization of the flow computed by the Boolean expression $b$, updated when the flow $a$ is true, starting with value $v$ :

$$i_0 = v, \qquad i_{n+1} = \begin{cases} b_n \text{ if } a_n \\ i_n \text{ otherwise} \end{cases}$$

In the very simple subset of DC we use here, a program is a tuple: $(I, O, L, D)$, where $I$ is the set of input flows, $O$ is the set of output flows, $L$ is the set of local flows, and $D$ is a set of definitions: $D = D^e \cup D^m$ is split into a set of EQU's $D^e$ and a set of MEMO's $D^m$.

# 4 From Behaviours to Boolean Equations

In the automaton framework, we have a clear distinction between *states* and *reactions*. The status of signals is meaningful during a reaction, while the states are meaningful at the beginning and at the end of a reaction. When we encode automata by equations, both states and signals will be encoded by Boolean flows, which take their values at each *instant*.

A reaction is a set of *enabled* transitions. A transition is said to be *enabled* iff its source state is active (i.e. the corresponding flow is true) at the beginning of the reaction and its guard is made true by the values of the input signals at the beginning of the reaction. The definitions define the next state by MEMO's, and the output signals by EQU's. Hence the outputs are simultaneous with the inputs that cause them, while state changes are effective only for the next reaction.

In the following subsections, we define the function $\mathcal{D}$ which associates a Boolean DC program to an ARGOS behaviour expression $E$: $\mathcal{D}(E) = (I, O, L, D)$. We shall describe the four components of this function separately: $\mathcal{D}_I(E)$, $\mathcal{D}_O(E)$, $\mathcal{D}_L(E)$, and $\mathcal{D}_D(E)$).

For simplicity, we use the signal and state names of the ARGOS program for naming flows in the associated DC program. This requires that we make some assumptions about the sets of names in the ARGOS behaviour expression, in order to avoid explicit renaming operations: let us suppose that all state and signal names are different in $E$.

## 4.1 Encoding Mealy machines

Let us first present a simple translation of Mealy machines into Boolean equations. This translation will be made a bit more complex in section 4.2.6 to cope with refinement operations. We associate a Boolean flow to each state. The $\mathcal{D}_D()$ function, applied to a Mealy machine $M = (S, s_0, I, O, T)$, gives a set of definitions in which we find MEMO definitions for states, and EQU definitions for the signals emitted by the automaton.

$$
\begin{aligned}
&\mathcal{D}_I((S, s_0, I, O, T)) = I\\
&\mathcal{D}_O((S, s_0, I, O, T)) = O\\
&\mathcal{D}_L((S, s_0, I, O, T)) = S\\
&\mathcal{D}_D((S, s_0, I, O, T)) = \{\mathcal{D}_D(s, T), s \in S\} \cup \{\mathcal{D}_D(o, T), o \in O\}
\end{aligned}
$$

State flows are *local* flows, in the sense of DC. Each state flow $s$ is defined by a *memo* definitions which depends on $T$: $\mathcal{D}_D(s, T) = \text{MEM}(s, rp, \text{TRUE}, \text{initv})$ where:

$$
rp = \left[ (s \wedge \neg \bigvee_{(s,b,-,-)\in T} b) \vee \bigvee_{(s',b,-,s)\in T} (s' \wedge b) \right]
$$

If the state is initial: $\text{initv} = \text{TRUE}$, otherwise $\text{initv} = \text{FALSE}$.

The idea is that, for the automaton to be in state $s$ at the end of the current reaction, there are two possibilities: either the automaton is already in state $s$ at the beginning of the current reaction, and it does not leave it, because the current value of the input signals is such that no transition sourced in $s$ can be taken; or the automaton is in a state $s'$, which has a transition to $s$, whose guard is verified by the current value of the input signals.

For correct (deterministic) Mealy machines, at each instant, there is a unique state flow whose value is true.

Each output signal $o$ is defined by an *equ* equation which depends on $T$: $\mathcal{D}_D(o,T) = \text{EQU}(o, rp, \text{TRUE})$ Where:

$$rp = \left[ \bigvee_{(s,b,O \ni o,-) \in T} (s \wedge b) \right]$$

The flow $o$ associated with a signal $o$ is true during the current reaction (i.e. at the start instant) if the signal belongs to the output set of an enabled transition (i.e. whose source state is active, and whose guard is verified). On the other hand, the right part of the equation is the constant FALSE if there is no transition emitting the signal.

## 4.2 Encoding behaviour expressions

### 4.2.1 Preliminary remarks

The encoding is syntax-directed: $\mathcal{D}(E_1 \ op \ E_2) = f_{op}(\mathcal{D}(E_1), \mathcal{D}(E_2))$.

For parallel composition, two components may emit the same signal, thus producing two equations for the same flow, which have to be merged.

In refinement, sub-behaviours are not always *active*. The flows needed to represent the behaviour of $E$ do not need to be computed at each step, if $E$ refines the state $A$ of a Mealy machine and this state is not always active

This is true for state flows. However, for the flows corresponding to signals, we must be very careful. In some sense, they *do have* to be computed at each step. If we do not compute the outputs of $E$ when $E$ is inactive, we obtain the following (wrong) behaviour: if the output signal $o$ has to be emitted at the instant when $E$ becomes inactive, then the corresponding flow is computed and set to TRUE. At the following instant, $E$ is inactive, and can no more participate to the emission of the signal. However, since we decided not to compute the flow any more, it remains TRUE, until the following activation of $E$ (see Figure 8-(a)).

However, the activation mechanism of DC is a way of generating efficient sequential code, since the flows do not need to be computed at each step. Hence, producing DC code that uses activation conditions as often as possible seems to be an important goal (although we should not forget that activation conditions are themselves expressions on flows, and that they have to be computed).

A possible compromise is the following: if a behaviour expression $E$ is to be embedded in a context where it is not always active, we may split the computation of an output flow $o$ into two parts: 1) $E$ is responsible for computing a partial "version" of the output flow, denoted by $o'$, and this can be done with an activation condition $a$, if $E$ is alive only when $a$ is true. 2) In the surrounding context, the correct definition of the output flow $o$ is given by: $\text{EQU}(o, a \wedge o', x)$, where $x$ is the activation condition of the context (see figure 8-(b)).

Hence the (possibly) complex expression that defines $o$ in the equations of a structure $E$ is computed only when this behaviour is active. We still need, however, to compute a simpler equation for the same output, in the context of $E$. The partial version $o'$ is used only when $a$ is true and all the complete versions of output flows (like $o$) have the same activation condition.

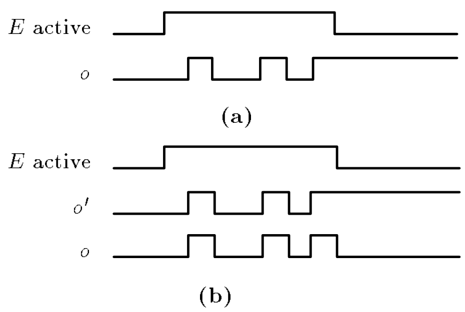


Figure 8: Partial and complete versions of output flows

#### 4.2.2 The *kill* and *alive* conditions

The idea is that each subexpression of the behaviour expression is encoded according to two additional inherited parameters $\mathsf{a}$ and $\mathsf{k}$. These parameters are the Boolean expressions that describe the *alive* and *kill* conditions of the given subexpression: *kill* means that the behaviour is killed in the current reaction (i.e. it is no more active at the end of the reaction); *alive* means that it is alive at the beginning of the current reaction (hence it participates to the current reaction). Note that $\mathsf{a} \wedge \mathsf{k}$ is possible.

The main call sets *alive* to TRUE and *kill* to FALSE, since the main behaviour is active at the beginning of all reactions, and is never killed. The *kill* and *alive* expressions are left unchanged when passed from a parallel node to its components. The same occurs for local signal declaration nodes. On the contrary, in a refinement operation, the kill and alive expressions of the operands depend on the structure of the machine that controls the refinement.

We define a new function $\mathcal{D}'_D$ with three parameters: a behaviour expression as before, and two Boolean expressions: the kill and alive conditions. The three other functions $\mathcal{D}_I$, $\mathcal{D}_O$ and $\mathcal{D}_L$ have the same parameters as before. For each expression $E$, the definitions in $\mathcal{D}(E, k, a)$ have the activation condition $a$.

#### 4.2.3 Main call

$\mathcal{D}_D(E) = \mathcal{D}'_D(E, \text{FALSE}, \text{TRUE})$

### 4.2.4 Parallel composition

We assumed that the sets of state flows of the components are pairwise disjoints. We simply take the union of the sets of MEMO equations that define states in all parallel components. Equations on signal flows are a bit more complex, since signals are shared by the parallel components. In particular, two components may emit the same signal, during the same reaction, or during different reactions. We would like to obtain a single equation defining each flow in a program. Hence for two equations that define the same flow, we take the *or* of their right hand parts.

If $\mathcal{D}'_D(E_1, k, a) = D_1^m \cup D_1^e$ and $\mathcal{D}'_D(E_2, k, a) = D_2^m \cup D_2^e$ (where the $D_i^m$ stand for the sets of MEMO equations and the $D_i^e$ for the sets of EQU equations), then

$$\mathcal{D}'_D(E_1 \| E_2, k, a) = D_1^m \cup D_2^m \cup \mathrm{M}ergeRightParts(D_1^e, D_2^e)$$

Consider the set $X = \{s \mid \exists r, a \text{ EQU}(s, r, a) \in D_1^e\} \cap \{s \mid \exists r, a \text{ EQU}(s, r, a) \in D_2^e\}$ of the multiply defined flows. Then:

$\mathrm{M}ergeRightParts(D_1^e, D_2^e) =$
$\{\text{EQU}(s, r, a) \in D_1^e | s \notin X\} \cup$
$\{\text{EQU}(s, r, a) \in D_2^e | s \notin X\} \cup$
$\{\text{EQU}(s, r^1 \vee r^2, a^1) | \text{EQU}(s, r^1, a^1) \in D_1^e \wedge \text{EQU}(s, r^2, a^2) \in D_2^e\}$

Notice that $a^1 = a^2$, by construction.

Finally:

$\mathcal{D}_I(E_1 \| E_2) = \mathcal{D}_I(E_1) \cup \mathcal{D}_I(E_2)$
$\mathcal{D}_O(E_1 \| E_2) = \mathcal{D}_O(E_1) \cup \mathcal{D}_O(E_2)$
$\mathcal{D}_L(E_1 \| E_2) = \mathcal{D}_L(E_1) \cup \mathcal{D}_L(E_2)$

### 4.2.5 Encapsulation

The equations of the encapsulated behaviour are left unchanged. The flows corresponding to the signals made local are *local* flows of the DC program.

$\mathcal{D}_D(\overline{E^\Gamma}, k, a) = \mathcal{D}_D(E, k, a)$

$\mathcal{D}_I(\overline{E^\Gamma}) = \mathcal{D}_I(E) - \Gamma,\ \mathcal{D}_O(\overline{E^\Gamma}) = \mathcal{D}_O(E) - \Gamma,\ \mathcal{D}_L(\overline{E^\Gamma}) = \mathcal{D}_L(E) \cup \Gamma$

### 4.2.6 Mealy machines (revisited)

Mealy machines are not behaviour expressions by themselves, but they can be encoded separately. The encoding below follows the same ideas as in section 4.1, and takes into account the *Kill* and *Alive* expressions inherited from the context.

$$\begin{aligned}
&\mathcal{D}_I((S, s_0, I, O, T)) = I \\
&\mathcal{D}_O((S, s_0, I, O, T)) = O \\
&\mathcal{D}_L((S, s_0, I, O, T)) = S \\
&\mathcal{D}_D((S, s_0, I, O, T), k, a) = \{\mathcal{D}_D(s, T, k, a), s \in S\} \cup \{\mathcal{D}_D(o, T, k, a), o \in O\}
\end{aligned}$$

Each state flow $s$ is defined by a *memo* equation: $\mathcal{D}_D(s, T, k, a) = \textsc{mem}(s, rp, a, \text{initv})$. If the state is initial: $\text{initv} = \textsc{true}$, otherwise $\text{initv} = \textsc{false}$. On the other hand, let us define, as before:

$$rp' = \left[(s \wedge \neg \bigvee_{(s,b,-,-)\in T} b) \vee \bigvee_{(s',b,-,s)\in T} (s' \wedge b)\right]$$

Then $rp = rp' \wedge \neg k$ for non-initial states, and $rp = rp' \vee k$ for the initial state.

This means that, when the machine is killed, we set it to its initial state (if $k$ is true, the flow corresponding to the initial state is true for the next reaction). It is harmless since state flows are not used when the machine is inactive, and it ensures that they will have the adequate value when the machine becomes active again.

All other states can be true only if $\neg k$, i.e., the machine is still active at the end of the current reaction. When a machine has never been active, the flow of its initial state is true, for the same reasons.

Each output signal $o$ is defined by an *equ* equation: $\mathcal{D}_D(o, T, k, a) = \textsc{equ}(o, rp, a)$ where:

$$rp = \left[\bigvee_{(s,b,O\ni o,-)\in T} (s \wedge b)\right]$$

We can also compute the *kill* and *alive* expressions to be transmitted to the refining behaviours, if the machine is used as the controller in a refinement operation:

$$\mathit{Alive}(s, T, k, a) = s \wedge a, \qquad \mathit{Kill}(s, T, k, a) = k \vee \bigvee_{(s,b,-,-)\in T} b$$

The behaviour refining the state $s$ will be active when $a$ (the controller itself is active) and $s$ are true; it will be killed when the state $s$ is left, i.e. a transition sourced in $s$ is enabled; or when the controller itself becomes inactive ($k$). Notice that $s$ may be true again at the end of the reaction, in which case the refining behaviour has been killed and simultaneously restarted in its initial state.

### 4.2.7 Refinement and NIL behaviours

For behaviours of the form $\mathbf{R}_M(R_1, ..., R_n)$, we need to compute partial versions of the output flows. We introduce the notation $D < o'/o >$: in the definitions of $D$, occurrences of $o$ are replaced by $o'$.

Let us note $M = (S = \{s_u | u \in [1, n]\}, s_0, I, O, T)$.

The encoding of the machine $M$ yields: $\forall u \in [1, n], k'_u = Kill(s_u, T, k, a), \quad a'_u = Alive(s_u, T, k, a)$. We compute the definitions of the controller: $D = \mathcal{D}_D(M, k, a)$ and the definitions of the refining behaviours: $\forall u \in [1, n], D_u = \mathcal{D}_D(R_u, k'_u, a'_u)$ with the suitable parameters *alive* and *kill*. Then:

$$
\begin{aligned}
&\mathcal{D}_I(\mathbf{R}_M(R_1, ..., R_n)) = I \cup \mathcal{D}_I(R_1) ... \cup \mathcal{D}_I(R_n)\\
&\mathcal{D}_O(\mathbf{R}_M(R_1, ..., R_n)) = O \cup \mathcal{D}_O(R_1) ... \cup \mathcal{D}_O(R_n) = \{o_j | j \in J\}\\
&\mathcal{D}_L(\mathbf{R}_M(R_1, ..., R_n)) = S \cup \mathcal{D}_L(R_1) \cup ... \cup \mathcal{D}_L(R_n) \cup \{o_j^u | j \in J, u \in [1, n]\}\\
&\mathcal{D}_D(\mathbf{R}_M(R_1, ..., R_n), k, a) =\\
&\qquad \bigcup_{u=1}^{u=n} D_u < o_j^u / o_j | j \in J > \; \cup D < o_j^0 / o_j | j \in J > \; \cup\\
&\qquad \{\text{EQU}(o_j, o_j^0 \;\vee\; \bigvee_{u=1}^{u=n} (a'_u \wedge o_j^u), a)\}_J
\end{aligned}
$$

For a signal $o_j$, possibly emitted by several refining behaviours and by the controller, there is one partial version per refining behaviour, denoted by $o_j^u$, and a partial version $o_j^0$ for the controller. The global equation of $o_j$ is computed with the activation condition $a$ and expresses that we have to take the correct value of $o_j$ from the controller or from the (unique) refining process which is currently active.

Finally: $\mathcal{D}_D(\text{NIL}, k, a) = \emptyset, \; \mathcal{D}_I(\text{NIL}) = \mathcal{D}_O(\text{NIL}) = \mathcal{D}_L(\text{NIL}) = \emptyset$

# 5 Example

For the system described figure 6, the set of DC definitions we obtain is:

$$
\begin{aligned}
&\text{MEM}(\mathsf{Running}, \mathsf{Running} \wedge \neg\mathsf{Stop} \vee \mathsf{Idle} \wedge \mathsf{Start}, \text{TRUE}, \text{FALSE})\\
&\text{MEM}(\mathsf{Idle}, \mathsf{Idle} \wedge \neg\mathsf{Start} \vee \mathsf{Running} \wedge \mathsf{Stop}, \text{TRUE}, \text{TRUE})\\
&\text{MEM}(s_1, s_1 \wedge \neg a \vee s_2 \wedge a \vee \mathsf{Stop}, \mathsf{Running}, \text{TRUE})\\
&\text{MEM}(s_2, s_2 \wedge \neg a \vee s_1 \wedge a, \mathsf{Running}, \text{FALSE})\\
&\text{MEM}(s_3, s_3 \wedge \neg\ell' \vee s_4 \wedge \ell' \vee \mathsf{Stop}, \mathsf{Running}, \text{TRUE})\\
&\text{MEM}(s_4, s_4 \wedge \neg\ell' \vee s_3 \wedge \ell', \mathsf{Running}, \text{FALSE})\\
&\text{EQU}(\ell', s_2 \wedge a, \mathsf{Running})\\
&\text{EQU}(A', s_4 \wedge \ell', \mathsf{Running})\\
&\text{EQU}(\ell, \mathsf{Running} \wedge \ell', \text{TRUE})\\
&\text{EQU}(A, \mathsf{Running} \wedge A', \text{TRUE})
\end{aligned}
$$

The initial values of $A$ and $\ell$ are irrelevant since these flows are always computed. The initial values of $\ell'$ and $A'$ are useful when the activation condition Running is false, but these flows are never used in this case.

## 6 Proving that the two semantics coincide

For a behaviour expression $E$, we can define: $\mathcal{S}(E) = (S, s_0, I_a, O_a, T) \in \mathcal{M}$ and $\mathcal{D}(E) = (I_d, O_d, L, D)$. Obviously, $I_a = I_d$ and $O_a = O_d$. We can compare $\mathcal{S}(E)$ and $\mathcal{D}(E)$ by considering that these two objects generate infinite sequences of $I \cup O \longrightarrow \mathcal{B}$, i.e. Boolean valuations of the relevant flows: inputs and outputs. If we note $\mathcal{T}_a$ (resp. $\mathcal{T}_d$) the function that associates the set of all such traces to a machine in $\mathcal{M}$ (resp. a DC program $(I_d, O_d, L, D)$), then we have to prove:

**Property 1 (The two semantics coincide)**
$\forall E \in \mathcal{E} \quad \mathcal{T}_a(\mathcal{S}(E)) = \mathcal{T}_d(\mathcal{D}(E))$

This means that the two objects we built from an ARGOS program are trace equivalent, for input/output traces. It is easier to show a stronger property: state behaviours over time also coincide. Hence we can prove that the two sets of traces are equal, even if we integrate states in the valuations. However, we should define precisely what we call states in the two objects, and how they can be compared.

**Correspondence of states and comparison of state valuations.** From the definition of the operations, each state $t$ of $\mathcal{S}(E) = (S, s_0, I, O, T)$ is of the form:

$$t ::= t \| t \ \mid \ s \triangleright r \qquad\qquad r ::= t \ \mid \ \text{NIL}$$

where $s$ is a state in a basic component of $E$. Let $BS(E)$ be the set of all basic states in $E$. Let $\mathit{Active}(t) \subseteq BS(E)$ be the set of those basic states that appear in a term $t$:

$$\begin{aligned}
&\mathit{Active}(t_1 \| t_2) = \mathit{Active}(t_1) \cup \mathit{Active}(t_2)\\
&\mathit{Active}(s \triangleright r) = \mathit{Active}(r) \cup \{s\}\\
&\mathit{Active}(\text{NIL}) = \emptyset
\end{aligned}$$

On the other hand, the states in $\mathcal{D}(E)$ are the flows defined by MEM's, we shall note it $F^m(E)$. A *global state* in the DC program is a Boolean valuation of the flows in $F^m$. Since $BS(E) = F^m(E)$ by construction, we can compare the two kinds of *global* states.

**Definition 1 (comparison of state valuations)** *We define a function $f$ from a state valuation of a Mealy machine $\mathcal{S}(E) = (S, s_0, I, O, T)$, of the form $S \longrightarrow \mathcal{B}$, to a state valuation of a* DC *program $\mathcal{D}(E) = (I, O, L, D)$ of the form $F^m \longrightarrow \mathcal{B}$ (where $F^m$ is the set of* MEM *defined flows). Let $w$ be a valuation: $S \longrightarrow \mathcal{B}$. Consider the unique $s \in S$ such that $w(s)$. Then:*

$$\begin{aligned}
&f(w)(a) = \text{TRUE}, \forall a \in \text{Active}(s)\\
&[f(w)(a) = \text{TRUE} \iff a \textit{ is initial in its machine}], \forall a \in BS(E) - \text{Active}(s)
\end{aligned}$$

*By construction $f$ is one-one.*

**Extracting input/output/state traces from a Mealy machine.** $\mathcal{T}_a((S, s_0, I, O, T))$ is defined as follows: a sequence $v_0, v_1, ...$ of valuations of the form $S \cup I \cup O \longrightarrow \mathcal{B}$ is a trace of the machine if and only if:

$$
\begin{aligned}
&(1) v_0(s_0) = \text{TRUE and } \forall s' \neq s_0 \; v_0(s') = \text{FALSE}\\
&(2) \; \forall k \geq 0, \exists ! s_c \in S \text{ such that } v_k(s_c) = \text{TRUE}\\
&\quad \text{and } \exists (s_c, b, o, s'_c) \in T \text{ such that}\\
&\quad v_k(b) \wedge v_k(o) = \text{TRUE and } v_{k+1}(s'_c) = \text{TRUE}
\end{aligned}
$$

**Extracting input/output/state traces from a DC program.** A sequence $v_0, v_1, ...$ of valuations of the form $I \cup O \cup L \longrightarrow \mathcal{B}$ is a trace of the DC program $(I, O, L, D)$ iff the properties that define EQU's and MEM's (cf. Section 3) hold at each instant of this sequence. We just have to extract inputs, outputs and MEM defined flows.

**Proof of property 1 (sketch).** It is sufficient to prove, for all behaviour expressions $E$, the conjunction of the two propositions below:

**(1)** The initial states coincide, i.e. all traces in $\mathcal{T}_a(\mathcal{S}(E)) \cup \mathcal{T}_d(\mathcal{D}(E))$ start with the same valuation (same values for inputs and outputs, values related by the function $f$ of the above definition, for states).

**(2)** A correct move according to the ARGOS semantics is a correct move according to the DC semantics, and vice-versa. Which means: if there is a finite trace $v_0, v_1, ..., v_n$ ($w_0, ..., w_n$ being the extracted state trace) in $\mathcal{T}_a(\mathcal{S}(E))$ and a trace $v'_0, v'_1, ..., v'_n$ ($w'_0, ..., w'_n$ being the extracted state trace) in $\mathcal{T}_d(\mathcal{D}(E))$, with $\forall i \in [0, n], w'_i = f(w_i)$, and $v_0, v_1, ..., v_n, v_{n+1}$ is a trace in $\mathcal{T}_a(\mathcal{S}(E))$ then $\exists v'_{n+1}$ such that $w'_{n+1} = f(w_{n+1})$ and $v'_0, v'_1, ..., v'_n, v'_{n+1}$ is also a trace in $\mathcal{T}_d(\mathcal{D}(E))$ ; and vice-versa.

(1) comes from the careful choice of initial values in MEM definitions; The proof of (2) is by induction on the structure of behaviour expressions.

# 7 Implementation and multi-language programming

The translation of ARGOS into DC is under development. We start from a structured definition of ARGOS, where one can define *processes.* A process has a formal interface (lists of input and output formal parameters), a list of local signals, and is described by a behaviour expression. A process can be *called* in the behaviour expression of another process, in which case formal inputs are instanciated by Boolean formulas on signals, and formal outputs are instanciated by signals. This gives ARGOS a structure similar to LUSTRE *nodes.* On the other hand, the complete DC format is also structured and provides the notion of *node.*

Translating both ARGOS processes and LUSTRE nodes into DC nodes, is a simple way of enabling the merge of ARGOS and LUSTRE programs. It is simpler than trying to translate ARGOS into LUSTRE directly, as we did in [JLMR94], since DC provides a notion of activation condition. Moreover, it allows the whole set of ARGOS programs to be considered, including the programs for which the direct translation into LUSTRE gives incorrect LUSTRE programs, because loops appear in the equations. For those programs, we also obtain loops in the definitions of the local flows. However, we can perform the loop detection and elimination operation described in [HM95] on the DC nodes obtained from ARGOS programs, before trying to merge them with other DC nodes.

Even if we are not interested in multi-language programming of reactive systems, the translation of ARGOS into DC gives a symbolic compilation technique for ARGOS. There is no size explosion in this first compilation phase: the size of the DC program is linear in the size of the ARGOS program. The second phase, which produces sequential code from a DC program, may be linear too, and is performed by a tool common to all synchronous languages. The notion of activation condition allows the generation of efficient sequential code, in which variables can be computed only when necessary.

Finally, we should point out that we used here a very simple subset of DC, whose complete version handles flows of any type, and deals with external objects such as procedures and functions from a host language.

# References


[Ber] G. Berry. *The constructive semantics of Esterel.* in preparation.

[Ber91] G. Berry. A hardware implementation of pure ESTEREL. In *ACM Workshop on Formal Methods in VLSI Design, Miami*, january 1991.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[CS95] C2A-SYNCHRON. The common format of synchronous languages – The declarative code DC version 1.0. Technical report, SYNCHRON project, October 1995.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[Har87] D. Harel. Statecharts : A visual approach to complex systems. *Science of Computer Programming*, 8:231–275, 1987.

[HM95] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *EUROMICRO*, Como, Italy, September 1995.

[IEE91] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), September 1991.

[JLMR94] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *In 5th IEEE International Conference on Computer Languages*, Toulouse, May 1994. IEEE Computer Society Press.

[JM94] M. Jourdan and F. Maraninchi. Studying synchronous communication mechanisms by abstractions. In *IFIP Working Conference on Programming Concepts, Methods and Calculi*, San Miniato, Italy, jun 1994. Elsevier Science Publishers.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. LNCS 630, Springer Verlag, august 1992.