

Argos: an Automaton-Based Synchronous Language

Florence Maraninchi^{1,4} Yann Rémond²

VERIMAG - Centre Equation, 2 Avenue de Vignate - F38610 GIERES³

Abstract

Argos belongs to the family of *synchronous languages*, designed for programming reactive systems (Lustre [1,2], Esterel [3], Signal [4], ...). Argos is a set of operators that allow to combine Boolean Mealy machines, in a compositional way. It takes its origin in Statecharts [5], but with the Argos operators, one can build only a subset of Statecharts, roughly those that do not make use of multi-level arrows. We explain the main motivations for the definition of Argos, and the main differences with Statecharts and their numerous semantics. We define the set of operators, give them a perfectly synchronous semantics in the sense of Esterel, and prove that it is compositional, with respect to the trace equivalence of Boolean Mealy machines.

We give an overview of the work related to the definition and implementation of Argos (code generation, connection to verification tools, introduction of non-determinism, etc.). This paper also gives a set of guidelines for building an automaton-based, Statechart-like, yet perfectly synchronous, language.

Key words: Argos, synchronous language, semantics, compositionality

1 Introduction

Reactive Systems and the Synchronous Approach

The term of *reactive system* [6] has been widely accepted to talk about computer systems in which the interactions with an environment are the promi-

¹ Institut National Polytechnique de Grenoble (INPG) and VERIMAG

² Université Joseph Fourier Grenoble (UJF) and VERIMAG

³ Verimag is a joint laboratory of UJF, CNRS and INPG

⁴ Corresponding author, Tel: (33) 4 76 63 48 53; fax: (33) 4 76 63 48 50.

E-mail address: Florence.Maraninchi@imag.fr

ment aspect. They are opposed to *transformational* ones, in which complex data structures and algorithms are involved.

The problem of *specifying, programming and verifying real-time reactive* systems, together with the definition of appropriate development environments, is still an important research problem [5–7], the following being widely accepted. There exist appropriate design methods, programming languages and environments for *transformational* systems (or, at least, for systems which are mainly transformational, like compilers). This is not yet the case for *reactive* systems, like real-time process controllers. Moreover, the need for formal verification methods and tools is even more crucial for reactive systems than for transformational ones, because strong reliability requirements are associated with them.

The family of *synchronous* languages [8] and formalisms has been a very important contribution to the domain. The synchronous approach is the mathematical foundation for the Esterel [3], Lustre [1,2], Signal [4] languages, for the algebra ATP [9], for the Concurrent Constraint Programming paradigm [10]. To a certain extent, some of the various semantics that have been proposed so far for Statecharts [5] are also synchronous. Modecharts [11,12] were recently given a synchronous semantics in the spirit of Esterel.

Synchronous languages or formalisms are based upon the *synchrony hypothesis*, which states that the reaction time of the system is zero. From an external point of view, it means that outputs are produced simultaneously with the inputs, which is clearly unimplementable; however, a synchronous system works fine provided it reacts sufficiently fast, w.r.t. the rate imposed by the environment: if relevant changes in the inputs occur at most each second, the system may take one second to react. When we program a reactive system using a synchronous language, we always have to prove that the final code can indeed execute one reaction of the system sufficiently fast. We would have to do that for any language. Simply, the structure of the typical imperative code produced by compiling a synchronous language is such that one can provide quite accurate upper-approximations of the execution time, looking at the source program. In the general case, computing the so-called *worst-case execution time* (see for instance [13]) of a program is difficult, mainly because of recursion and unbounded loops; but these constructs never appear in the code of a synchronous program.

The interesting part is the *internal* point of view on the synchrony hypothesis. The hypothesis states that the reaction time of a particular component, and the communication time between components, are also zero. This makes the semantics of parallel composition very simple. It could be useless if the implementation of the parallel composition in synchronous languages was based on a separate implementation of each component, running in parallel with a kind

of dynamic scheduler, because the hypothesis that the communication takes no time would clearly be in contradiction with the reality of the execution. But the synchronous languages are intended to be compiled into *centralized sequential code*. The parallel composition and the communication mechanism are introduced only for *description* purposes, at the level of the language; they are compiled into something sequential, hence they do not imply explicit parallelism and communication at execution time. (The problem of *distributing* code is also important, but can be considered orthogonal to the fact that there indeed exists a parallel structure in the source language; see work by P. Caspi [14,15]).

The synchrony hypothesis implies that systems compose very well, and are easier to describe and verify than asynchronous ones. Moreover, synchrony allows to deal with functional and timing correctness of a system separately [7,16].

Finally, synchronous languages are *programming* languages, not only specification languages. Their programming environments provide efficient compilers towards various kinds of software or hardware targets. Relying on a formally defined semantics also allows the connection of such programming languages to validation tools (formal verification by model-checking or deductive methods, test case generation, debugging, etc.). For Lustre, see for instance [17,18].

Argos

The first definition of Argos appeared in [19,20]. Argos is a set of operators for combining Mealy machines in parallel and hierarchic structures; it has a pure synchronous semantics following that of Esterel; it may be given a graphical syntax similar to that of Statecharts (with no multi-level arrows), and is indeed inspired by the very first papers on Statecharts [5] and Higraphs [21]. It is listed in [22] as one of the Statecharts semantic variants, but it was not designed for that purpose, and a lot of Statecharts features are missing. We show, however, that some of these features may be described from the primitive set of Argos operators, thanks to the synchronous semantics. We give below the main motivations for the definition of Argos.

No multi-level arrows

The main difference between Argos and the various semantics of Statecharts in which people have tried to get rid of the multi-level arrows, is that the perfectly synchronous semantics we adopted allows to replace this rather *syntactical* feature by a completely *semantical* one: some processes, at two different levels of the hierarchy, do communicate with each other and this ensures a global behavior similar to that of a multi-level arrow. In some sense, this essential

modification is similar to what happened when explicit `goto`'s were replaced by implicit ones, available through the use of semantical structures like `WHILE` loops. The important point is that one cannot do without something replacing multi-level arrows: they correspond to very usual situations in the reactive system domain (or you are also bound to forget about hierarchy of states!).

The choice we made for Argos has big consequences: with no multi-level arrows, a Statechart-like picture may be seen as a particular combination of well-defined automata. An automaton is a set of states and transitions connected together, and this is possible only at one level. Then complex pictures are obtained by two simple constructs: put two (possibly composed) objects in parallel; put a (possibly composed) object inside the state of an automaton. This paves the way to a well-defined *syntax* of programs, on which a structural semantics can be based. When a syntax-directed semantics has been obtained, compositionality begins to make sense, and a careful definition of the composition operators guarantees it.

A Compositional semantics

What does compositionality mean, in this context? In all programming languages, one can build several syntactically different programs that *do the same*. When we are in the process of defining a formal semantics, this notion may (should) be formalized a little. For Argos, it means the following: the semantics of a program is given in terms of a Boolean Mealy machine, the mathematical model we use for reactive behaviors. There exists an equivalence relation for such machines, that captures the fact that two machines represent the *same* reactive behavior (something similar to the fact that, in a sequential language, `while true do A od` and `while true do A; A od` are the same).

Knowing this equivalence relation, we can define the compositionality criterion for the semantics of Argos: if two sub-programs are the same (i.e. the semantics associates with them two equivalent machines), then one should be able to replace one by the other *in any program context*, without changing the semantics of the global program (formally: the equivalence is a congruence for all the operators of the language).

We show that this is true for the semantics of our operators set. To our opinion, compositionality is a key property for the language to be usable: it allows to reason about sub-programs independently.

A notion of incorrect programs

A less prominent (but yet important for programming real systems) difference between Argos and all the Statechart semantic variants we know is our notion

of *incorrect program*. We will see in the sequel that the communication mechanism adopted in Statecharts and Argos gives rise to the so-called *causality* problems, somewhat similar to deadlocks. Imagine a process that waits for signal *a* for emitting signal *b*, talking to a process in parallel that waits for *b* for emitting *a*: what should the behavior of the whole be? Are the two processes stuck, waiting for each other to start? Or does a kind of spontaneous reaction take place? Let us call the programs in which such problems appear “*non-causal*”.

For Statecharts, people tried to give a meaning to *all* combinations of objects, in particular non-causal ones. This leads to some choices in the semantics that may have consequences on the compositionality properties, for instance. Huizing [23] has studied the relationships between what he calls *responsiveness* (obeying the synchrony hypothesis), *causality* and *modularity* (a notion related to our compositionality), and proved, in his framework, that no semantics can be responsive, causal and modular. See more details in section 6.3.

In Argos, we followed the path shown by Esterel: there are *incorrect* compositions, to which we are not bound to give a meaning. As a consequence, we are interested in compositionality properties for the subset of *correct* programs only. This is much simpler than trying to integrate a notion of compositionality with a way of giving a meaning to non-causal programs; moreover, it is legitimate: this is the point of view of a *programming language* designer. In all programming languages one can write incorrect programs that are detected at compile-time and rejected: no meaning is given. This can be due to typing, for instance.

Of course, we should provide a detection mechanism for non-causal objects. This is a bit more complex than typing in classical languages, however, especially for valued (i.e. not only Boolean) Argos or Esterel. This is because the *exact* detection of such incorrect programs is undecidable. For the Boolean subsets of the languages, it is decidable, but may be quite expensive, because it depends on the expanded control structure of the program.

In classical languages, the detection of a large class of errors is also undecidable. A few errors only can be detected at compile-time (like $X := 1/0$; in Ada) ; otherwise the sources of potential dynamic errors (like `get(Y) ; X := 1/Y ;`) are clearly identified, and the compiler produces “defensive” code, which may raise exceptions at execution-time. Hence, the static mechanism may accept incorrect programs, but it provides well-defined dynamic errors.

In the family of synchronous languages, dedicated to the programming of safety-critical reactive systems, we do not accept dynamic errors. Hence we have to adopt the opposite point of view, and to provide conservative *ap-*

proximate detection mechanisms: if a program is accepted by the detection mechanism, then it is guaranteed to be free of non-causal situations (i.e., dynamic errors); if it is rejected, it may contain non-causal situations, or it may be free of these problems too. When such a language is augmented with arrays or other data structures, the same principle applies: we cannot accept an “index out of bounds” error at execution-time, hence we are bound to be quite drastic at compile-time.

The quality of the approximate detection mechanism is good if it does not reject correct programs too often.

A small number of features (or constructs)

Finally, we deliberately gave priority to a clean and simple semantics, and did not hesitate to reject some features, if they did not fit well in the simple semantic framework. It appears now that a lot of sophisticated language features (some of them borrowed from Statecharts) can be described as macro-notations, using a very simple core language. We mention some of them in section 6.1. This is good news for code-generation, connection to analysis tools, etc.

Outline of the Paper

The paper is organized as follows: first, in section 2, we present the notion of a Boolean reactive system, for which the basic Argos operator set is designed, we show a simple example that uses the three main operators of Argos, and we explain their semantics intuitively. Section 3 defines the set of operators formally. Section 4 defines a language based upon this set of operators, and the notion of causally incorrect program. Section 5 discusses code generation issues, and the connection to analysis tools. Section 6 gives a (probably non-exhaustive) list of related work, with detailed comparison. Section 7 is the conclusion.

2 An Argos Example with intuitive Semantics

2.1 Boolean Reactive Systems

Boolean Mealy machines constitute the basic components of Argos programs. They are appropriate for the description of *Boolean reactive systems*, i.e. reactive systems in which the inputs and outputs are *pure* signals. A digital watch

may be seen as a Boolean reactive system: each button of its interface gives one Boolean input signal, and we can use two Boolean outputs *on* and *off* for each element of the digital display. It makes a huge set of outputs (the display of a single digit needs 7 elements, hence 14 signals), but there is no loss of information.

The digital watch may also be seen as a reactive system with integer outputs, in which case the physical environment is supposed to deal with outputs like 10, in order to control the display. In some sense, this only changes the position of the frontier between what we call the *system*, to be described in our language, and what we call the *environment*.

This is usually sufficient for event systems, like the digital watch. For describing signal-processing systems, or simply control systems in which the inputs are given by sampling a continuous phenomenon, one needs *valued signals*, e.g., integers, reals, etc. Although most complex reactive systems have both event-driven and sampled subsystems, the core of Argos deals with pure signals only, encoded into Booleans. Values are introduced in section 4.5.

2.2 Description of the program

Figure 1-a is an Argos program using four automata (or Boolean Mealy machines), to describe a modulo-8 a-counter, with initialization and interruption facilities. Rounded-corner boxes are automaton states; arrows are transitions; rectangular boxes are used for unary operators (see below). A set of states and transitions which are connected together constitutes an automaton. The four basic components of the program have the following sets of states: {Counting, Not counting}, {A0, A1}, {B0, B1}, {C0, C1}.

In an automaton, transitions are labeled by inputs and outputs. The input part of a transition label is a Boolean formula of the input signals, not necessarily a complete monomial (the *end* label stands for $\text{end.stop} \vee \text{end.}\overline{\text{stop}}$); the output part is a set of output signals. (The input part and the output part are separated by a slash; negation is denoted by over-lining, and conjunction is denoted by a dot: $c/\text{end}, \text{stop.}\overline{\text{end}}$. When the output set is empty, it can be omitted). We also omit the transitions from a state to itself, if they do not emit signals. For instance, the states A0 and A1 of the first bit (resp. B0 and B1, C0 and C1) should have loops labeled by \overline{a} (resp. $\overline{b}, \overline{c}$).

There is one initial state, designated by an arrow without source. States are named, but names should be considered as comments: they cannot be referred to in other components. An arrow can have several labels — and stand for several transitions, in which case the labels are separated by a comma.

The automaton whose states are **Counting** and **Not counting** is said to be *refined*, in its **Counting** state, by a subprogram built with the three other automata. The external box, whose cartridge contains **end**, is the graphical syntax for a local signal declaration unary operator. The box defines the scope in which the signal **end** is known. This signal is used as input by the refined automaton; it is used as output by one of the three other ones: a communication will take place between the two. Another such unary operator is used in the program, in order to limit the scope of signals **b**, **c** to the program constituted by the three unrefined automata.

The interface of the global program is defined as follows: all signals which appear in a left-hand (resp. right-hand) side of a label, and are not declared to be local to some part of the program are *global inputs* (resp. *global outputs*). The shadowed box gives the name of the program (or subprogram), the list of global inputs, and the list of global outputs.

Finally, three automata are put *in parallel*: they are drawn separated by dashed lines.

2.3 Intuitive Semantics

The behavior is as follows.

Initially, the counter is not counting; the global state is **Not counting**. It may be started by the input **start**, which puts the system in the global state **Counting:A0B0C0**, encoding the value 0. The first occurrence of the signal **a** then moves the first bit from **A0** to **A1** and leaves the two other bits unchanged: **Counting:A1B0C0** encodes 1. The next occurrence of **a** moves the first bit back to **A0**; this transition is labeled by **a/b**, which means that it broadcasts the signal **b** towards the other components, that may react to it in the same reaction; it moves the second bit from **B0** to **B1**. The third bit is left unchanged: **Counting:A0B1C0** encodes 2. The fourth occurrence of **a** moves the first bit from **A1** to **A0**, which emits **b**; hence it also moves the second bit from **B1** to **B0**, which emits **c**; hence it moves the third bit from **C0** to **C1**. The global target state is **Counting:C1A0B0**, encoding 4.

The eighth occurrence of the signal **a** moves the counter from state **Counting:A1B1C1** to state **Counting:A0B0C0**, and emits the signal **end**, which is an input of the main automaton: the system returns to the global state **Not Counting**.

At any moment, a **stop** signal stops the counter, and the system returns to the global state **Not Counting**. The label **stop. \overline{end}** is necessary for ensuring the *determinism* of the main automaton (see details below). However, since

the two transitions labeled by $\text{stop}.\overline{\text{end}}$ and end do not emit signals, and have the same target state, they could be replaced by a single one, labeled by $\text{stop} \vee \text{end}$.

2.4 Equivalent programs

The second program (Figure 1-b) is *equivalent* to the first one (see formal definition below): the parallel composition of the three bits, with the signals b and c being declared local, has been replaced by a flat automaton, having the same behavior.

Since the semantics is *compositional*, replacing a component by an equivalent one leaves the global program behavior unchanged.

Figure 2 is a third equivalent program, made of one single flat automaton. The semantics of the language formally defines the translation of the first program (Figure 1-a) into the flat automaton of Figure 2.

3 A set of operations on Boolean Mealy machines

In this section, we first define the objects that serve as basic components in all Argos programs: Boolean Mealy Machines, with additional properties like determinism and reactivity.

Then we define a set of *operations* on these objects that may, or may not, preserve the additional properties.

This set of operations is not yet a *language*. In the language, we add a syntax for programs (composed objects, i.e. expressions made of basic objects and any number of operators), and define the notion of *correction* of a program.

The definition of the operations makes them *total*, but the notion of program correctness may declare that a particular combination of basic objects and operators is illegal, in which case we are not bound to give it a meaning.

3.1 Deterministic and Reactive Boolean Mealy Machines

A simple reactive behavior may be described by a *labeled transition system*. The transition system has one *initial* state. Transition labels are made of two parts: the *input* part I , and the *output* part O . The complete label is denoted

by I/O. Both parts are built upon a set of elementary interactions with the environment, called *signals*.

The input part is a Boolean condition on signals. It describe a condition to be fulfilled by the environment in order to make the system react. A Boolean condition describes a *set* of input signal valuations.

The output part is the set of signals the system outputs to its environment, when reacting to a given input. One transition is one reaction, and is supposed to be instantaneous, hence the outputs are simultaneous to the inputs that cause them. Time passes in states.

Let \mathcal{A} denote the set of *signals*. In the general case, a basic Argos component is of the following form :

Definition 1 : Boolean Mealy machine

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 . \square

Without loss of generality, we can always consider that the Boolean Mealy machines we deal with have only complete monomials as input labels. If the set of inputs is $\{a, b\}$, then the input condition a stands for $a \wedge b \vee a \wedge \neg b$, and a transition labeled by a/o may be split into two transitions (between the same states) labeled by $a \wedge b/o$ and $a \wedge \neg b/o$.

Definition 2 : Bisimulation of Boolean Mealy machines

Two machines $M_1 = (S_1, s_{01}, I, O, T_1)$ and $M_2 = (S_2, s_{02}, I, O, T_2)$ are said to be *bisimilar*, denoted by $M_1 \approx M_2$, if and only if there exists an equivalence relation $\mathcal{R} \subseteq S_1 \times S_2$ such that $s_{01} \mathcal{R} s_{02}$ and

$$s \mathcal{R} s' \implies \left\{ \begin{array}{l} s \xrightarrow{b/o} r \implies \left\{ \begin{array}{l} \exists b_1, \dots, b_m, r'_1, \dots, r'_m \text{ such that} \\ \forall i \in [1, m], s' \xrightarrow{b_i/o} r'_i \wedge r \mathcal{R} r'_i \\ \wedge b \implies (\forall_i b_i) \end{array} \right. \\ \text{and conversely.} \end{array} \right.$$

\square

Bisimulation has been first introduced by Park and Milner [24,25]. It coincides

with trace equivalence for deterministic systems.

The machines we consider should be both deterministic and reactive. We give the definition only for machines labeled by complete monomials on the set of inputs.

Definition 3 : Determinism and reactivity

A machine (S, s_0, I, O, T) is *reactive* iff :

$$\forall s \in S, \left[\bigvee_{(s,m,o,s') \in T} m \right] = \text{true}$$

It is *deterministic* iff :

$$\forall s \in S, \forall t_1 = (s, m_1/o_1, s_1) \in T, \forall t_2 = (s, m_2/o_2, s_2) \in T \\ m_1 = m_2 \implies (o_1 = o_2) \wedge (s_1 = s_2)$$

□

We will denote by \mathcal{M}^r , \mathcal{M}^d and \mathcal{M}^{rd} , respectively, the sets of *reactive*, *deterministic*, *reactive and deterministic* Boolean Mealy machines.

Determinism is an important issue. In spite of the inherent non-determinism in the description of the environment, the programs should describe deterministic behaviors. In this framework, non-determinism of a reactive behavior is simply the existence of two transitions sourced in the same state, with non-exclusive input parts, and different output parts and/or target states.

3.2 Operations

3.2.1 Cartesian Product or Parallel Composition

The formal definition of parallel composition is based upon the following product operation.

Definition 4 : Synchronous product of Boolean Mealy machines

$$\begin{aligned}
\times & : \mathcal{M} \times \mathcal{M} \longrightarrow \mathcal{M} \\
(S_1, s_{01}, I_1, O_1, T_1) \times (S_2, s_{02}, I_2, O_2, T_2) & = \\
(S_1 \times S_2, (s_{01}, s_{02}), I_1 \cup I_2, O_1 \cup O_2, T') &
\end{aligned}$$

Where T' is defined by :

$$\begin{aligned}
((s_1, m_1, o_1, s'_1) \in T_1, (s_2, m_2, o_2, s'_2) \in T_2) & \implies \\
((s_1, s_2), m_1 \wedge m_2, o_1 \cup o_2, (s'_1, s'_2)) & \in T'
\end{aligned}$$

□

The synchronous product of Boolean Mealy machines is both commutative and associative, and it is easy to show that it preserves both determinism and reactivity.

Note that the parallel composition does not make any synchronization between components. It is the appropriate construct for the parallel composition of two *independent* systems. When the systems have to communicate or synchronize with each other, parallel composition should be used together with encapsulation of some dedicated signals; this is explained below.

Since all the components are *reactive*, a transition in the composed process corresponds to exactly *one* transition in each of its parallel components. Some of them execute loops, and emit no signals, so their reaction is not observable, but they do take a transition, and only one.

The machine corresponding to the parallel composition of the two first bits of the counter (before applying the unary operator that declares \mathbf{b} and \mathbf{c} to be local) is given in Figure 3.

3.2.2 Encapsulation

Basic ideas

Encapsulation is a unary operator parameterized by a set of signal names. It is used to restrict the scope of signals, and to force synchronization between parallel or hierarchic components. Typically, if a signal s is used as the output of a component P and as the input of a component Q , it may serve as a synchronization signal. This is the case for signals b , c and \mathbf{end} of the counter

in section 2. The synchronization and communication mechanism is the *synchronous broadcast* (the same as in Esterel): the sender can always send, and it needs not know whether 0, 1 or several other components are listening this signal. Sending is non-blocking.

The main reason why we express the semantics of the synchronous broadcast only in the encapsulation operator is the following: it may serve for synchronizing parallel components, but also hierarchical components. If we partly integrate it in the semantics of the parallel composition, then we need to repeat it in the semantics of the hierarchic composition.

Observation of the example

The intuitive semantics of the example shows that the bit in which b is an input should react to this signal only if it comes from the previous bit. On the other hand, the signal b is emitted by the first bit in order to synchronize with the second one, and should not be visible elsewhere. In such a case, the scope of b can be restricted to the parallel composition of the first and second bits. In the example, it is in fact extended to the parallel composition of the three bits, but it does not appear in the third one, and it is simpler to use only one encapsulation operator for the two signals b and c .

Defining the scope of a signal by encapsulating a subprogram P , allows to simplify the transitions of P : encapsulation forces the synchronization between the components of P by removing some transitions of their product, like the restriction in CCS [26]. It is a bit more complex because of the input/output structure of the labels, and the Boolean structure of the inputs, but it is essentially the same idea.

Formal definition

Definition 5 : Encapsulation

$$\backslash : \mathcal{M} \times 2^A \longrightarrow \mathcal{M}$$

$$(S, s_0, I, O, T) \backslash \Gamma = (S, s_0, I \backslash \Gamma, O \backslash \Gamma, T')$$

Where T' is defined by :

$$\begin{aligned} (s, m, o, s') \in T \wedge m^+ \cap \Gamma \subseteq o \wedge m^- \cap \Gamma \cap o = \emptyset \\ \implies (s, \exists \Gamma . m, o \backslash \Gamma, s') \in T' \end{aligned}$$

□

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

Intuitively, a transition $(s, m, o, s') \in T$ is still present in the result of the encapsulation operation if its label satisfies the local criterion: $m^+ \cap \Gamma \subseteq o$, which means that a local signal which is supposed to be present has to be emitted in the same reaction; and $m^- \cap \Gamma \cap o = \emptyset$, which means that a local signal that is supposed to be absent should *not* be emitted in the same reaction.

If the label of a transition satisfies this criterion, then the names of the encapsulated signals are hidden, both in the input part and in the output part. This is expressed by $\exists \Gamma.m$ for the input part, and by $o \setminus \Gamma$ for the output part.

The encapsulation operator can only remove some transitions in a complex object obtained, for instance, as the result of a parallel composition. Hence it is always true that each basic automaton component in a program participates in a global reaction by executing exactly *one* transition. The synchronization based on broadcasting signals does not give rise to infinite behaviors, or lack of stability, etc.

The synchrony hypothesis, stating that communication takes no time, is illustrated here: if we consider two programs P and Q , communicating with a signal a which is made local to their parallel composition, one transition in P that emits a , and one transition in Q that reacts to the presence a , make a *single* transition in the result. The parallel composition is completely *compiled*, and there is nothing like a communication at execution time.

Determinism and Reactivity of an encapsulated process

The encapsulation operation does not preserve determinism nor reactivity. Intuitively, this is because the criterion used for ruling out some transitions of the encapsulated process, depending on their labels, is applied *locally*.

Take a component P with two states A_1 and A_2 and four transitions $(A_1, i \wedge a/b, A_2)$, $(A_1, \bar{i} \wedge a, A_1)$, $(A_1, \bar{i} \wedge \bar{a}, A_1)$, $(A_1, i \wedge \bar{a}, A_1)$, and a component Q with two states B_1 and B_2 and four transitions $(B_1, i \wedge b/a, B_2)$, $(B_1, \bar{i} \wedge b, B_1)$, $(B_1, i \wedge \bar{b}, B_1)$, $(B_1, \bar{i} \wedge \bar{b}, B_1)$.

Put these two components in parallel and make the signals a and b local to the result. For input i , from the composed state A_1B_1 , there will be two distinct transitions left when applying the above criterion: (A_1B_1, i, A_2B_2) and (A_1B_1, i, A_1B_1) . The first one is made of: the transition $(A_1, i \wedge a/b, A_2)$ in P ,

and the transition $(B_1, i \wedge b/a, B_2)$ in Q , which gives $(A_1B_1, i \wedge a \wedge b/a, b, A_2B_2)$ in the product, before encapsulation; it passes the local criterion and then gives (A_1B_1, i, A_2B_2) when hiding a and b . The second one is made of the transition $(A_1, i \wedge \bar{a}, A_1)$ in P and the transition $(B_1, i \wedge \bar{b}, B_1)$ in Q , which gives $(A_1B_1, i \wedge \bar{a} \wedge \bar{b}/\emptyset, A_1B_1)$ in the product, before encapsulation. It also passes the local criterion, and then gives (A_1B_1, i, A_1B_1) when hiding a and b .

The result is no longer deterministic, because of these two transitions. There are no other transitions left in the encapsulated program.

Now, take the same component P , and a component Q with two states B_1 and B_2 and four transitions $(B_1, i \wedge \bar{b}/a, B_2)$, $(B_1, i \wedge b, B_1)$, $(B_1, \bar{i} \wedge b, B_1)$, $(B_1, \bar{i} \wedge \bar{b}, B_1)$.

Put them in parallel and encapsulate the result with local signals a and b : in the resulting process, there is *no* transition sourced in A_1B_1 for input i : the result is no longer reactive.

These are the typical cases where non-determinism and non-reactivity appear. It may seem strange to write such pathological communications directly, in such simple systems. But this kind of situation may appear in very complex compositions of components, with any number of participants.

Alternative view of the semantics

Another way of expressing the semantics of the encapsulation operator is by giving a system of equations, of which the values of the encapsulated signals are a solution. This view gives some new hints for understanding why determinism and reactivity are not preserved.

The idea is the following: for each state q of the process P , and each configuration I of the inputs, there should be exactly one *status* of the encapsulated signals, i.e. one valuation of these signals, seen as Boolean variables. Knowing a configuration of the inputs, and a status of the local signals, it is easy to determine the reaction of the system, by observing what transitions can indeed be taken.

The status of the local signals is the solution of a set of equations that can be built as follows. For each local signal s , take the set of transitions sourced in q and emitting s (i.e. with a label of the form: c_i/S with $s \in S$) and build the equation: $s = \bigvee c_i$. In other words, s is true (emitted) if and only if at least one of the transitions of P that emits it can be fired. This expresses the fact that a local signal cannot come from outside P : it is present in a reaction of P only if P itself emits it. The important point is that the local signals may appear in the conditions c_i . Hence the system of equations obtained by writing

such an equation for each local signal is of the general form; in particular, it may contain cycles of dependencies, like $a = b; b = a$.

When the system of equations has exactly one solution, it means that the reaction of the system to input I is unique. If this is the case for each state of P , then the encapsulated system is both deterministic and reactive.

If, for a state q , the system of equations has more than one solution (resp. no solution at all) then the encapsulated process is no longer deterministic (resp. reactive). This is the case for processes that exhibit equations of the form $a = b; b = a$ (several solutions) or $a = b; b = \bar{a}$ (no solutions).

The typical examples presented above give this kind of system: $a = i \wedge b; b = i \wedge a$, or $a = i \wedge \bar{b}; b = i \wedge a$. Since we are interested in the reaction of the system to input i , we replace i by `true` in these equations, and obtain the typical cases.

The two-bit example

Figure 4 shows how to apply the encapsulation operator, for signal b , to the system we obtained in figure 3.

An Example: the Instantaneous Dialogue

Observe the example of figure 6. It shows an *instantaneous dialogue* between the two parallel components. The intuitive behavior is the following: when in state A , the first component P_1 needs to query the state of the second component P_2 in order to choose its reaction to input i ; if P_2 is in state C , then P_1 goes to B , otherwise it stays in A . (In Statecharts, this would be described by a condition of the form “ i and $in(C)$ ” appearing in P_1 . We explain in section 4.4 below why we chose not to introduce such a construct in Argos).

In Argos, P_1 reacts to I by emitting Q , which is a question to P_2 . P_2 reacts to this question by saying yes (Y) if it is indeed in state C , and nothing if it is in another state, say D , or E . The answer and the question are simultaneous, thanks to the synchronous broadcast. Hence the behavior of P_1 , from state A , is described by: if I and Y , then emit the question Q and take the transition to B ; if I and not Y , then emit the question Q and stay in A . This may seem strange, because there is no visible sequencing between the question and the answer, but the behavior of the global process, where both the question Q and the answer Y are local signals, is exactly what we want. This is because the status of Q and Y , when the system is in the global state AC , is given by the equations: $Q = i \wedge Y \vee i \wedge \neg Y; Y = Q$, which simplifies to $Q = Y = i$. Hence

for input i , both Q and Y are present, the global system takes the transition from A to B and the loop on C .

Notice that, in Esterel, from which this example is borrowed, the notion of sequence *inside a reaction* allows a natural writing of the dialogue. The program in figure 5 is made of two parallel components. The first one emits the question Q when the cause I occurs. Then it checks whether the answer Y is present, for emitting the output S (we need something observable, like changing state in the Argos program). The second parallel component always answers Y when it receives the question Q (the answer could depend on some internal state, of course, as in the Argos example). `await tick`, meaning “wait until next instant” is necessary because the component `every Q do emit Y` does not emit Y if Q is present in the very first instant, but only after that. The parallel composition does nothing in the first instant, and then behaves as expected.

In section 3.2.4 below, we give another version of this program by using the Argos refinement operator, which is more natural. In Esterel, there is even a third solution, in which P_2 emits Y continuously, when in the appropriate state, not only when it receives the question Q . In this case P_1 needs only test Y , without asking Q .

3.2.3 Inhibiting Operator

The inhibiting operator is useful for building some of the Statecharts or Esterel constructs from the set of Argos operators, in a structural way (see section 3.3 below). P `whennot` a behaves as P , but only when a is not present. The interface of P `whennot` a has one more input signal.

Definition 6 : Inhibition of a Boolean Mealy machine

$$\begin{aligned} \text{whennot} & : \mathcal{M} \times \mathcal{A} \longrightarrow \mathcal{M} \\ (S, s_0, I, O, T) \text{ whennot } a & \text{ is defined iff } a \notin I \text{ and} \\ (S, s_0, I, O, T) \text{ whennot } a & = (S, s_0, I \cup \{a\}, O, T') \end{aligned}$$

Where T' is defined by :

$$(s, m, o, s') \in T, \implies (s, m.\bar{a}, o, s') \in T' \quad \wedge \quad (s, m.a, \emptyset, s) \in T'$$

□

3.2.4 Hierarchical Composition

The intuitive behavior of a refined process is the following: a transition that enters a refined state starts the refining process, in its initial state. A transition that leaves a refined state kills the refining process, and all information about the state it had reached is lost (this, in particular, forbids the “entry by history” of Statecharts).

An important point is the following: the automaton that is refined is always active; when it is in state A , the process refining state A is also active. They behave as if they were put in parallel, as far as the communication between them is concerned. But, of course, a transition of the refined automaton may kill a refining process and start another one, which is not the case for parallel composition.

Note that the encapsulation operator can be applied to the result of a hierarchical composition too, for synchronizing a controller with its refining processes.

Finally, a refining process is alive during the reaction that kills it (leaving a refined state is a non-preemptive interrupt). Hence it may communicate (or synchronize) with the refined automaton. In particular, the transition of the automaton that leaves the state may be triggered by a signal emitted by the refining process itself. In this case, we say that the refining process commits suicide. We will see that this particular feature of Argos allows to encode outgoing multi-level arrows in an elegant and compositional way. This is the case in the introductory example for the signal *end*, which is emitted by the three-bit counter, and kills it. Conversely, in the example, the signal *stop* acts as an external interrupt, whatever the state of the three-bit counter is.

On the contrary, the process that refines the *target* state of a transition occurring in the refined automaton is not alive during this reaction: it does not participate in the reaction that starts it. This forbids a number of interesting behaviors (choosing the actual initial state by an initial transition triggered by external signals, ...) and the symmetrical encoding of ingoing multi-level arrows. We could modify the semantics of the refinement operator in order to make it more symmetrical. However, the basic version we present here is simpler, because it does not give rise to the *schizophrenia* problem of Esterel (see [27] for details). Roughly speaking, schizophrenia would occur for a loop on a refined state: the refining process is both killed and restarted in its initial state. In fact, two instances of it would be alive during the reaction, not necessarily in the same state. If the refining process contains local signals, they all have two instances during one reaction.

With our asymmetrical semantics, schizophrenia cannot occur.

The formal definition is based upon the operation denoted by \triangleright , that takes

a machine for the controller and n machines for refining the n states of the controller. The machine that serves as the controller is not necessarily reactive.

Definition 7 : Refinement

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

Let M denote (S, s_0, I, O, T) , where $S = \{s_0, s_1, \dots, s_n\}$. Consider also a set $\{M_j\}_{j=0..n}$ of machines, to be used as refinements of the states of M , where $M_j = (S^j, s_0^j, I^j, O^j, T^j)$, and $S^j = \{s_0^j, s_1^j, \dots, s_{n_j}^j\}$.

The composed machine, in which each M_j refines the state s_j , is of the form:

$$M \triangleright \{M_j\}_{j=0..n} = (S \triangleright \{S^j\}_J, s_0 \triangleright s_0^0, I \cup \cup I^j, O \cup \cup O^j, T').$$

Its set of states is of the form:

$$S \triangleright \{S^j\}_J = \cup_{j=0}^n \{s_j \triangleright s_k^j, k \in [0..n_j]\}$$

And its transitions T' are given by the following two 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 machine that refines the current state. The outputs are gathered. In the global target state, the machine that refines state s_b is started in its initial state, and the machine that refines state s_a has been killed. Its internal state is no longer relevant:

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

- 2) A transition of the machine that refines the current state, from s_k^a to $s_{k'}^a$, while no transition sourced in state s_a is activated in the controller:

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

□

Figure 7 illustrates the semantics of the hierarchical composition (or “refinement”).

Figure 8 is a version of the instantaneous dialogue using refinement. The transitions labeled by i/Q and by Y in the first component may be taken

together, if the other component is in state C and indeed answers Y to the question Q . When the system is in state AX , the status of the local signals Q and Y is given by the equations: $Q = i; Y = Q$, which gives $Q = Y = i$ as in the parallel version of figure 6, without the need for Boolean simplification.

3.3 Some Useful Combinations of Operators

3.3.1 Temporized states

The ability to attach *delays* to states was proposed in an early unpublished paper about Statecharts. In ARGOS, it is very easy to introduce such a construct and to give it a clear semantics; indeed, it can be introduced as a *macro-notation*. Following synchronous languages (Lustre, Esterel, Signal), ARGOS deals with *multiform time*. Any external input event can be used as a clock for the system, which may count meters as well as seconds. Figure 9-a shows a *temporized state* **Tempo**. The notation $[d \ a]$ means that a delay d is associated with the state. d is a positive integer, and a is the name of an input signal, which gives the *unit of time*. The behavior of the system is as follows: when the system enters state **Tempo**, it can stay in this state at most until d occurrences of signal a have been counted; if it has not left the state when the last occurrence happens, the special transition denoted by a box (the *time-out transition*) is taken. Figure 9-b shows how the macro-notation is expanded: states are added to count the occurrences of a .

All transitions that enter the temporized state output the starting signal **start**; all transitions that leave the state output the killing signal **kill**; the box transition is triggered by the **time-out** signal. The main point is the priority we choose between the killing transitions and the counting ones. In the expansion we give, an outgoing transition can be taken even if the last occurrence of a occurs at the same time (**time-out** is output only if **kill** does not occur).

If we add *variables* to Argos (i.e. objects that exist “in parallel” with the Argos program, and that may be tested and assigned to during a reaction), the counter is described by a variable, which avoids explicit states. However, as far as verification is concerned, we have to expand these variables into explicit states, for using model-checkers, if we want to verify properties in which time is involved. In [28] we showed how to translate Argos programs with temporized-states directly into *timed-graphs* [29] (without expanding the macro-notation into explicit states), in order to use the verification tool Kronos [30]. Some examples using the Argos compiler connected to Kronos are described in [31,32].

3.3.2 Inhibiting transitions

When the automaton which controls a refinement operation takes a transition, the subprogram which refines the source state always reacts at the instant when it is killed.

When preemptive interruptions have to be described, one uses *inhibiting transitions*, which stand for inhibiting operators (see figure 10). The transition sourced in **State 1** (figure 10-a) outputs the signal which inhibits the refining subprogram P: it is a preemptive interruption; the macro-notation avoids the introduction of α and uses a small black circle (figure 10-b).

4 The Argos Language and its Semantics

4.1 Syntax

From the set of operators described in the previous sections, we define the core of the Argos language. \mathcal{E} is the set of programs :

$E ::= E \parallel E$	Parallel composition
$\overline{E^\Gamma}$	Encapsulation $\Gamma \subseteq \mathcal{A}$
$\mathbf{R}_M(R_1, \dots, R_n)$	refinement of M by the R_i
$\overline{E^{<\gamma>}}$	Inhibition $\gamma \in \mathcal{A}$
$R ::= E \mid \text{NIL}$	refining objects

The NIL notation is introduced to identify leaf states, i.e. the states that are not further refined.

4.2 Causality and Incorrect Compositions

The term “*causality*” has been widely accepted, following the authors of Esterel, to talk about those situations when the strict interpretation of the synchrony hypothesis leads to apparent paradoxes. When a process that waits for signal **a** for emitting signal **b**, is talking to a process in parallel that waits for **b** for emitting **a**, the global behavior is not defined. This typical example, written in Argos, is given in section 3.2.2 above.

In Argos, we decide to *characterize* a causally incorrect composition of components (or *program*) by the fact that there exists an occurrence of an encapsulation operator which is applied to a reactive and deterministic component and yet yields a system which is either non-deterministic or non-reactive. These are the cases when non-determinism (or non-reactivity) *appears*, due to the application of the encapsulation semantics, based on the definition of the synchronous broadcast mechanism.

Hence our notion of correction is the following: all basic components should be both deterministic and reactive, and all operators should be applied in such a way that they preserve these properties. We are convinced that the programs that will be rejected by our criterion indeed constitute programming errors. This notion of program correction is, in some sense, minimal, with respect to the synchronous broadcast mechanism; it is also called *logical correctness*. See section 6.3 for comparisons with similar notions in other synchronous languages and Statecharts.

Under these constraints, we can easily associate a flat Boolean Mealy machine to any Argos program, and this is indeed what we describe in the semantics below. Then this machine can easily be implemented by translation into any imperative programming language like C, Ada, Java, etc.

Any other choice, i.e. accepting to take non-deterministic or non-reactive Boolean Mealy machines as the semantics of Argos programs, would be unimplementable.

For non-reactivity, consider a state X , from which the transition for input i is missing: there is no consistent implementation. Notice that an implementation of the system that does nothing for input i when in state X is in fact an implementation of the system that does have a loop on state X , with input i , and no emitted signal. (In section 3.2.2 above, we showed that it may be the case that even the loop does not exist).

For non-determinism, there is no implementation, unless we consider that flipping a coin at execution time, for choosing a transition, is an appropriate solution.

However, we could find non-determinism useful, in a *specification* language, especially for the partial description of the environment. In this case, we would allow the basic components to exhibit some explicit non-determinism (several transitions with non-exclusive input conditions, and distinct emitted signals and/or target states), but still require that the composition of two such components *do not introduce* non-determinism. This is what we argue in [33].

All these remarks lead to the notion of incorrect program in Argos. Incorrect programs should be detected, of course, by a compiler.

In [34] we proposed an exact detection mechanism for causality errors, for Argos with pure signals; it gives a reasonable cost compilation algorithm.

4.3 Semantics

Since there exist incorrect programs, the semantic function should be partial on \mathcal{E} . We make it total by adding the special value \perp to the codomain.

The semantic function $\mathcal{S} : \mathcal{E} \rightarrow \mathcal{M}^{rd} \cup \{\perp\}$ is defined recursively by:

$$\mathcal{S}(E_1 \parallel E_2) = \begin{cases} \perp & \text{if } \mathcal{S}(E_1) = \perp \text{ or } \mathcal{S}(E_2) = \perp \\ \mathcal{S}(E_1) \times \mathcal{S}(E_2) & \text{otherwise} \end{cases}$$

$$\mathcal{S}(\mathbf{R}_{M^d}(R_1, \dots, R_n)) = \begin{cases} \perp & \text{if } \exists i \in [1, n] \text{ s.t. } \mathcal{S}(R_i) = \perp \\ M^d \triangleright (\mathcal{S}(R_1), \dots, \mathcal{S}(R_n)) & \text{otherwise} \end{cases}$$

$$\mathcal{S}(\overline{E^\Gamma}) = \begin{cases} \perp & \text{if } \mathcal{S}(E) = \perp \\ \text{otherwise: let } X = \mathcal{S}(E) \setminus \Gamma \text{ in} & \text{if } X \in \mathcal{M}^{rd} \text{ then } X \text{ else } \perp \end{cases}$$

$$\mathcal{S}(\overline{E^{<\gamma>}}) = \begin{cases} \perp & \text{if } \mathcal{S}(E) = \perp \\ \mathcal{S}(E) \text{ whennot } \gamma & \text{otherwise} \end{cases}$$

$$\mathcal{S}(\text{NIL}) = (\{\text{NIL}\}, \text{NIL}, \emptyset, \emptyset, \{(\text{NIL}, \text{true}, \emptyset, \text{NIL})\})$$

A program P is said to be *incorrect* if and only if $\mathcal{S}(P) = \perp$. The errors are due to encapsulations that do not preserve reactivity or determinism. The \perp value is absorbant.

4.4 Compositionality

From the definition of the equivalence for Boolean Mealy machines, we define an equivalence of Argos programs, denoted by \equiv . The main point here is that we are interested in compositionality for correct programs only.

Definition 8 : Equivalence of Argos programs

$$P_1 \equiv P_2 \iff \begin{cases} \mathcal{S}(P_1) \neq \perp \wedge \mathcal{S}(P_2) \neq \perp \wedge \mathcal{S}(P_1) \approx \mathcal{S}(P_2) & \vee \\ \mathcal{S}(P_1) = \mathcal{S}(P_2) = \perp & \end{cases}$$

□

The semantics is compositional, which means that the equivalence of Argos programs is a congruence for the operators (parallel and hierarchic compositions, inhibition, encapsulation):

$$\forall P, Q \in \mathcal{P}, \forall \mathcal{C} \text{ context} \quad P \equiv Q \implies \mathcal{C}[P] \equiv \mathcal{C}[Q]$$

It is easy to prove, by induction on the structure of processes (see [20]). (We can include the proof in the full paper if needed).

Remark

Since we require the equivalence of Boolean Mealy machines to be a congruence for our operators, and since this equivalence does not take state information into account, we cannot use state information in the semantics of our constructs.

Let us take an example: the equivalence defined above is in fact a kind of trace equivalence, which may identify two machines with different sets of states, provided they have the same paths. For instance, the machine with two states A and B , and four transitions $(A, a/x, B)$, (A, \bar{a}, A) , $(B, a/x, A)$, (B, \bar{a}, B) is equivalent to a machine with only one state C and two transitions $(C, a/x, C)$, (C, \bar{a}, C) . With our semantics, the first machine may be replaced by the second one (or the other way round) in any context, without changing the behavior of the global program in which that occurs.

This forbids, in particular, to give a direct semantics to the Statechart feature in which one may write “**a and in (A)**” as a condition for a transition to be taken, where A is the name of a state, somewhere in a component of the program. Indeed, the component with state A could be replaced by an equivalent one with no state A . However, it is relatively easy to replace the syntactic feature “**in(A)**” by a communication based on exchanging a dedicated signal.

4.5 Introducing variables

The Boolean core of Argos is now completely defined. We mentioned the need for *variables* or *valued signals* previously. Variables could serve as counters for avoiding explicit states, for instance in the encoding of temporized states. Valued signals are needed for representing the inputs and outputs of regulation systems, in which the computer samples continuous data, like the temperature.

We never implemented a complete Argos with variables, but the ideas for introducing are quite simple, and we explain them briefly here.

4.5.1 Boolean Mealy machines with Variables

In Argos, we can introduce variables by upgrading Boolean Mealy machines to general interpreted automata.

We still have a set \mathcal{A} of pure *signals*, i.e. Booleans. An Argos component with variables is now of the following form :

Definition 9 : Boolean Mealy machine with variables

A Mealy machine with variables is a tuple (S, s_0, I, O, V, T) where $I \subseteq \mathcal{A}, O \subseteq \mathcal{A}$ are the sets of input and output signals; \mathcal{V} is the set of (potentially typed) *variables* used in this machine, taking their values in a domain \mathcal{D} ; S is the set of states; s_0 is the initial state; $T \subseteq S \times \text{cond}(V) \times \mathcal{B}(I) \times 2^O \times \text{Assign}(V) \times S$ is the set of transitions. \square

As before, $\mathcal{B}(I)$ denotes the set of Boolean formulas with variables in I .

$\text{cond}(V)$ is the set of Boolean conditions on V . For instance, if V contains an integer x and a Boolean b , " $x < 0 \wedge \neg b$ " is a possible condition.

$\text{Assign}(V)$ is the set of assignments to variables in V . For instance, with the same hypothesis as before, " $\{x := 2.7; b := \text{false}\}$ " is a possible assignment. The elementary assignments are considered to be in parallel : there should not be two assignments to the same variable in the same set, but there is no order.

The intuitive idea is that a transition (q, c, m, o, a, q') , where c is the condition on variables, m is a condition on Boolean inputs, o is the set of emitted signals, and a is an assignment, is taken if: the automaton is in q , m is true of the external inputs; and c is true in the current valuation $\sigma : V \rightarrow \mathcal{D}$ of the variables. The automaton goes to state q' , emitting the signals in o , and the variables are updated according to σ and a . This gives a new valuation σ' .

A lot of things become undecidable when variables are introduced (especially integers, with the power of full arithmetics. In fact, interpreted automata have the power of Turing machines: it is quite easy to convince oneself that they can be used as a target language for any high level programming language).

For instance, it becomes impossible to check the determinism of the basic components, in the general case. There are several ways the problem can be solved: we can introduce a syntactic feature that allows to specify *priorities* between the transitions sourced in the same state. From the semantic point of view, this is like requiring the exclusivity of the conditions to be statically checkable, i.e. to have the following form: $c_1, c_2 \wedge \neg C_1, c_3 \wedge \neg(c_1 \vee c_2)$, etc.

Similarly, reactivity is not checkable but, on each state A of the basic components, we can add a loop that emits nothing and leaves the variables unchanged, labeled by $\neg \bigvee c_i$, where the c_i are the conditions of all the explicitly given transitions sourced in A . If the automaton is already reactive, this condition reduces to **false**, and has no influence on the behavior.

4.5.2 Compositions of Boolean Mealy machines with Variables

We can rewrite the semantic rules, taking the valuations of variables into account. In the parallel composition, a combined transition is labeled by: the conjunction of the Boolean conditions on inputs (as before), the conjunction of the conditions on variables, the union of the emitted signals (as before), the union of the assignments. We declare an error if this results in assigning to a variable twice. Hence, if a variable is shared by n parallel components, one should be the producer (the one allowed to assign values to the variables) and all the others should be only consumers (allowed to read the variable).

When we introduce variables, we should also introduce a unary operator to define their *scope*. Such an operator may be parameterized by: the name of the variable, its type, and an initial value. If a variable is declared local to a process that refines the state A of an automaton, it is reinitialized each time state A is entered.

4.5.3 Extending the notion of incorrect program

The main problem concerns the detection of so-called *causality errors*. Indeed, the mechanism we presented is based upon the *existence* of transitions for a given input, in the result of an encapsulation. When variables are introduced, it might be the case that a transition that remains in the result of an encapsulation is in fact not firable, because of its condition on variables. Take, for instance, a component P with a transition labeled by $(\mathbf{x} < 0)$ a/b and a component Q with a transition labeled by $(\mathbf{x} > 0)$ b. If we put them in

parallel and encapsulate the signal \mathbf{b} , we obtain a transition labeled by $(\mathbf{x} < 0) \wedge (\mathbf{x} > 0) \mathbf{a}$, which is clearly non-firable.

Should we decide that there is *no transition* for input \mathbf{a} , in the result of the encapsulation and, consequently, declare a case of non-reactivity? Let us call *intrinsic correctness* the notion of correctness based upon the existence of transitions, with the interpretation of conditions being taken into account.

Even if we chose this new definition of correct programs for Argos with variables, we could not implement it: the problem is undecidable. We cannot compute statically the set of transitions that are indeed firable. The general problem is even more complex because it may depend on the dynamics of the system: think of a transition labeled by $\mathbf{X} < 0$, sourced in a state that cannot be entered unless $\mathbf{X} > 0$.

Since an *exact* detection mechanism cannot be defined, we should be able to define an *approximate* one.

There is a way of providing a *conservative detection mechanism* for both determinism and reactivity: we consider each condition on variables as a new Boolean input. A transition label of the form: $\mathbf{C}(\mathbf{x}) \mathbf{m}/\mathbf{o}$, where $\mathbf{C}(\mathbf{x})$ is an arbitrary Boolean condition on the variable \mathbf{x} , is treated as $\alpha \wedge \mathbf{m}/\mathbf{o}$, where α is a fresh signal name (not used elsewhere in the program). We can do that on all transitions of the basic components of a program. Of course, we lose a lot of information: $\mathbf{x} < 0$ and $\mathbf{x} \geq 0$ will be replaced by two independent Boolean variables (or signals). Then, we apply our static criterion to this new program. The criterion requires that there exist exactly one possible transition for each configuration of the inputs (including the ones introduced for encoding the conditions).

We may reject intrinsically correct programs, of course, because the detection mechanism may complain about non-determinism or non-reactivity appearing for a given input $\alpha \wedge \beta \wedge \dots$, where α and β are new names introduced as explained before for representing conditions c_1 and c_2 , and $c_1 \wedge c_2$ is in fact not satisfiable. But we cannot accept intrinsically incorrect programs.

Hence we have a conservative detection mechanism. Moreover, we are convinced that it does not reject “too many” correct programs. This is a rather informal statement, but it means the following: the programs that are rejected, while being intrinsically correct, are those in which the correctness relies on some intricate mixing of Boolean Signals with other variables. Writing such programs is a questionable practise, because the slightest modification may transform an intrinsically correct program into an incorrect one, or conversely.

4.5.4 Valued signals

If variables are available, a valued signal is simply a pair made of a pure signal that represents the presence of the signal, and a variable that contains its value. This is the approach of Esterel, which is particularly well suited for event systems. In Lustre, there is no notion of a pure signal: all inputs have values, of type `int`, or `real`, or `bool`.

One can then offer a specific syntax for designating the presence of a signal and its value. In Esterel, we would write `x` for the presence (hence `present x then . . .` is correct) and `x?` for the associated value. (However, in Esterel, be careful not to confuse *valued signals* with *variables*).

5 Code Generation and Connection to Analysis Tools

Argos programs may be compiled into a lot of automaton formats, used as input by verification tools.

For producing explicit automata, a compiler that mimics the definition of the operators would be far too expensive. Indeed, the intermediate objects obtained when expanding parallel compositions are likely to be far bigger than the final system obtained by applying encapsulation operators for all local signals. We use a top-down method, with BDDs [35] for solving the equations obtained for the encapsulation operations. This is described in [36]. The Argonaute environment based upon Argos (a graphical editor and simulator, plus a compiler) has been successfully connected to various verification tools, among which: Aldebaran [37], Mec [38], Kronos [30–32], Polka [39,40].

It is easy to obtain executable code from an explicit flat automaton corresponding to an Argos program; the typical form of such a sequential program is an infinite loop:

```
initialize the state
while true
  get inputs (the values of the input signals)
  compute outputs, according to the values
                        of the inputs and the current state
  emit outputs
  update the state
```

Each pass in the loop corresponds to one *transition* of the global Mealy machine obtained by expanding the Argos program, hence it also corresponds to one *reaction* of this program. A reaction usually involves several components.

For the synchrony hypothesis to be a usable approximation of the real world, the execution time of one pass in the loop should be less than the minimal amount of time between two relevant changes of the program environment.

However, the compilation into an explicit automata is not always a good idea, since it produces code whose size is exponential in the size of the program (the parallel composition, in particular, produces an explosion of the number of states). For generating good sequential code, Argos is compiled into data-flow equations with activation conditions, using the DC format [41], also used as an intermediate form in the compilers of Lustre, Signal and Esterel [42]. DC can then be compiled into C. We also obtain a program with an infinite loop like the one above, but the set of states is not given in extension. Rather, the state is the configuration of a set of Boolean variables, which may be assigned to separately.

6 General comments and Comparison with Related Work

First, we summarize how to provide some of the missing Statecharts features as macro-notations in Argos. Then we compare Argos with three classes of Statecharts semantics. The way communication, synchronization and the associated “causality” problems are treated is somewhat independent of these three classes. We review the main choices in a separate section.

6.1 From Argos to Statecharts

Outgoing multi-level arrows can be done thanks to an explicit communication between the refining process P and the automaton A it refines, since P participates in a reaction in which the state A is left.

Ingoing multi-level arrows can be done with a similar mechanism if we modify the definition of refinement in such a way that the process that refines a state participates in a reaction in which this state is entered. We did not present this extended version of the refinement here, because it gives rise to schizophrenia problems. See [43] for an example of use.

Special events like "`entered(A)`" can be implemented as macro-notations, with pre-processing: we introduce a new signal name, and add it to the set of emitted signals of all transitions that enter A . This has to be done carefully, because A may be the initial state in a refining process, and there is no explicitly drawn transition that enters it. But then the signal can be added to the set of emitted signals of the transition that starts this process. Similar things

can be done for the special events "`exit(A)`" and "`in(A)`".

Adding variables and valued signals has been described in section 4.5.

Entry by history cannot be done, at least as a simple macro-notation, because the definition of the refinement is entirely based upon the fact that all information about a process refining a state is lost when we exit the state. However, if we look at systems in which entry by history is used, we understand that what we really need is a *suspension* mechanism. Suspension may be built in Argos with a simple combination of parallelism, inhibition and communication, as it is shown in Figure 11. The interface signals are `suspendP` and `resumeP`. We may decide whether suspension has an immediate effect (i.e., P does not participate in the reaction that suspends it). In the figure, this is the case, since the transition labeled by `suspendP` does indeed emit the signal α which inhibits P . We should also take care of the simultaneous occurrence of `suspendP` and `resumeP`.

This construction is quite satisfactory because, if we need entry by history in a process P , it means we need P to stay alive, until we need the information about its internal state again. And, if it is still alive, it is intrinsically *in parallel* with another part of the program. In Argos, we use refinement only when we need to start and kill processes, depending on a sequential structure described by an explicit automaton. In all situations where we need to keep some information on a process, even if it does not participate in the reactions for a while, we use the parallel composition.

6.2 Comparison with other Statecharts semantics

As far as we know, the semantics that have been proposed so far for Statecharts or Statecharts variants⁵ fall into three main categories. This classification is somewhat biased, since we are mainly interested in the use of Statecharts as a programming language; the existence of a notion of equivalence — or congruence — is an important criterion. The set of references is by no means exhaustive: a quick world-wide-web search gives at least 150 references on Statecharts, and a thorough comparison of all the variants is not the subject of this article.

In order to compare Argos and Statecharts, the following three classes are

⁵ We find semantics of Statecharts in a wide variety of papers, since the formalism has been used intensively in a number of very different contexts. As soon as a translation of Statecharts into some formally defined language is provided, we can consider that a semantics of Statecharts is given.

adequate:

- *Global* semantics
- *Denotational* and fully abstract semantics
- *Process Algebraic* semantics, with equivalences or congruences

They are detailed below.

None of the variants we know of has been especially tailored for pure *programming* purposes, and this is a major difference with Argos. All of them are presented as *specification* languages, that can be used in a verification framework. Sometimes, an algorithm for generating code is provided but, in all cases, generating code implies taking a decision about the non-deterministic elements of the language.

6.2.1 *Global Semantics of Statecharts*

In this class, a Statechart is defined as an And/Or tree, with additional constraints on the structure of the tree. The semantics is described in terms of a large number of tree-manipulation functions, like the closest common ancestor of two nodes, etc. [44–46] and a lot of others fall into this category. There is no way to view a program as a composition of sub-programs, and there is no real “syntax” — or grammar, on which a syntax-directed semantics could be based. Therefore, a Statechart is viewed as a monolithic object, not as the composition of simpler objects, and compositionality makes no sense. Another consequence is that none of these semantics propose a notion of *equivalence* for Statecharts. The operationally-defined semantics like that of [47], which describes the algorithm of the interpreter, also falls into this class.

6.2.2 *Denotational Fully Abstract Semantics of Statecharts*

In this class, the language is given a syntax, and a syntax-directed semantics. When the full set of features of Statecharts is indeed taken into account (like the multi-level arrows in [48]), the combinators are quite complex, and sometimes far from “semantical” combinations. One need to build programs by composing basic objects that can be sets of states with dangling incoming or outgoing transitions, and it is hard to attach a meaning to these objects, in terms of reactive behaviors. In this class of semantics, the notion of compositionality relies on the full abstraction criterion.

6.2.3 Process Algebraic Semantics of Statecharts

In this class, the language is also given a syntax, in the spirit of process algebras: the classical non-deterministic choice (the “+” of CCS [26]) and the parallel operator are used. [49–51] and many others fall into this class.

These semantics also need a notion of *model* for Statecharts, i.e. the mathematical definition of a reactive system behavior (usually a labeled transition system, or LTS). Giving an operational and process-algebraic semantics to Statecharts means providing a structural translation of Statecharts into a LTS. The main differences with Argos are the following.

First, the construction of programs starts with the prefix operator. Basic automata are built with the prefix operator, the non-deterministic choice, and the recursion. Then, automata may be composed using the parallel operator, for instance. Multi-level transitions are often forbidden, for the same reasons as in Argos, but without explaining clearly how a similar behavior can be obtained with the features of the language. In fact, since the underlying semantics is not purely synchronous, there is no solution with a macro-notation, i.e. independent of the context in which it is needed.

With this granularity in the definition of the basic components, these semantic frameworks also need to require *well-guardedness* (see the definition of this notion in CCS [26]) for avoiding infinite branching. In Argos, we consider automata as the primitive objects, because it is easier to define the determinism and reactivity criteria in such a framework. Hence we need no recursion operator.

Second, Argos is a subset of Statecharts features, except on one point: we decided, from the very beginning, to define an *encapsulation* operator, that allows to restrict the scope of signals. This has well-known renaming effects, which are necessary when describing large systems (think of a version of C without local variables, in which a team of 50 developers would have to decide who is allowed to use *i* as a loop index!). But this has also some advantages concerning the definition of the language itself: the encapsulation operator is the one that corresponds to the synchronization. As already mentioned, it is the same idea as in CCS.

6.3 Communication, synchronization and causality

6.3.1 Causality in other synchronous languages

As we illustrated with the definition of encapsulation, the synchrony hypothesis means that the status of *local* signals, used for internal synchronization,

should be uniquely determined in a global transition: it is given as the solution of a system of equations. The intrinsic paradoxes are related to the fact that the system of equations may have 0 or several solutions. This is called *logical correctness*.

In Esterel, the notion of causally correct program is much stronger, because we do not only require that the system have a unique solution, but also that this solution be *constructive* (see [16] for a synthetic presentation of this idea: the system of equations has to be solved using constructive logic, i.e. without using $a \vee \neg a = 1$).

In Lustre, the data-flow declarative style of the language means that “programs” are similar to the set of equations we showed in section 3.2.2 for explaining encapsulation. The static criterion used for ruling out non-causal programs is even stronger than in Esterel: the system of equations should be cycle-free. A cyclic system of equations, even if it has a unique solution, is rejected. It may seem too strong a criterion, but there are very few cases in which we would like the criterion to be a little weaker (see examples, taken from circuit design, in [34,52]).

6.3.2 Notions related to causality in Statecharts

To our knowledge, none of the semantics that have been proposed so far for Statecharts is purely synchronous, in the sense that: they are based upon the synchrony hypothesis, and there is a notion of incorrect program for ruling out the programs that rise the causality paradoxes intrinsic to the strict application of this hypothesis.

By *strict application of the synchrony hypothesis*, we mean that a component that contains a transition labeled by a/b , in parallel with a component that contains a transition labeled by b , gives a *single* transition as a result, whatever the notion of “result” is. For process-algebraic semantics, this is the same notion as in Argos, since they are based on operations over LTS. For global semantics, this is also quite similar, because the operational semantics describes the possible global transitions of a Statechart. Another point of view on the same fact is the following: a/b in parallel with b giving a single transition in the parallel process means that the internal communication between the two components is *atomic*, with respect to what happens in the environment.

This global transition is computed knowing the status of *local* signals used for internal synchronization, which should be uniquely determined.

In Statecharts, there is no real notion of *incorrect* program: non-determinism is not considered harmful, and if a transition is missing (non-reactivity), the system simply does nothing. In general, there is no notion of *local* signal either,

for which we could require a unique status. Hence the notions of *causality* are quite different from the corresponding notions in synchronous languages.

The very first semantics of Statecharts ([45] for instance) introduced the notion of “*micro-step*”, to talk about what happens inside a Statechart when it reacts to an external stimulus. There is a complementary notion of *macro-step*, intended to represent an atomic reaction of the system to its environment; a micro-step is usually a *maximal* sequence of micro-steps, in the sense that it cannot be extended: none of the currently active components can react to the signals that were present in the original external stimulus, or that were emitted by the already executed micro-steps. This raises new problems, because a maximal sequence does not necessarily exist in all cases; in general, it is not possible to determine statically whether all reactions to external inputs give finite sequences of internal micro-steps. Moreover, in some of the so-called micro-step semantics, it is possible to take one micro-step assuming that a signal is absent and, later, to take another micro-step that emits the signal. This kind of problem led to a number of criteria like *global consistency*, for talking about those macro-steps that are logically correct, when observed globally.

A good summary can be found in [23], where it is shown that no micro-step semantics can be *responsive*, *causal*, and *modular*. Responsiveness means that the outputs are simultaneous with the inputs (the synchrony hypothesis); modularity means that a component may be used in an overall context knowing its macro-step semantics *only* and forgetting about the details of its micro-steps (this is related to our compositionality criterion, with a notion of equivalence defined as having the same macro-steps) ; causality means that a macro-step cannot generate its own trigger (this is one half of our definition: it rules out programs of the form: a/b in parallel with b/a).

The semantics of Statecharts entitled “what is in a step?” [53,44] is sufficiently close to ours for the comparison to be meaningful: we did it in [54], where we proposed a general framework for comparing synchronous communication mechanisms; the differences lie in the interpretation of the synchronous broadcast. For instance, the parallel version of the instantaneous dialogue (figure 6) would be correct in this version of Statecharts, but would give no behavior. The refinement version of the dialogue works fine, though. In our semantics, where these two programs are indeed *equivalent*, there cannot be such a semantical difference between them.

7 Conclusions

We presented a set of operators for building a synchronous language based on explicit automata, with a syntax similar to that of Statecharts.

The semantics is purely synchronous; it may be viewed as a simplification of the Esterel semantics, adapted to a programming style where automata are given explicitly (in Esterel, the control structure of a program is made of loops, watchdogs, etc.).

As it is, the set of operators we presented is not a *language*, because it lacks a lot of features of real programming languages, for instance valued (i.e. not only Boolean) signals, explicit variables of any type that can be tested and assigned during a reaction, etc. Moreover, the other synchronous languages allow to use calls to a host language like C, in which complex data structures may be defined. This should be the case for Argos too.

In order to provide such features in a simple way, and also because we are convinced that complex systems cannot be described using only automata, we worked a lot on the combination of Argos with other synchronous languages (mainly Lustre and Esterel, which have quite distinct programming styles: Lustre is data-flow, while Esterel is imperative with control structures). See for instance [55]. We also worked on the Argos-Esterel combined language, but Charles André and his team (University of Nice, France) have defined the SyncCharts [56] formalism, which seems to be *the* way things should be done: the language inherits various preemption primitives from Esterel, the hierarchic program structure from Argos, and other interesting primitives from Grafset [57], which may be given a readable and concise graphical semantics.

More recently, Argos has given birth to Mode-Automata [58], a promising mixed-style language for the description of regulation systems and their running modes, that can be completely implemented on top of the Lustre both academic and commercial programming environments. A basic mode-automaton is an automaton with data-flow programs attached to states. The language re-uses the parallel and hierarchic constructs of Argos, although the basic objects are more similar to Moore machines than to Mealy machines. As in Argos, there are no multi-level arrows in the hierarchic composition, and the components at several levels of hierarchy may communicate with each other in order to achieve the multi-level transition behavior.

Acknowledgements

The authors wish to thank G. Berry, for numerous discussions and guidance during the definition of Argos; N. Halbwachs for the Lustre point of view; M. Jourdan for all her work on Argos (part of the compiler, the connection to Kronos, and the multi-language approach); and Ph. Schaar for his work on the graphical interface.

References

- [1] P. Caspi, N. Halbwachs, D. Pilaud, J. Plaice, LUSTRE, a declarative language for programming synchronous systems, in: 14th Symposium on Principles of Programming Languages, Munich, 1987.
- [2] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous dataflow programming language lustre, *Proceedings of the IEEE* 79 (9) (1991) 1305–1320.
- [3] G. Berry, G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation, *Science Of Computer Programming* 19 (2) (1992) 87–152.
- [4] P. L. Guernic, A. Benveniste, P. Bournai, T. Gauthier, Signal: A data flow oriented language for signal processing, Tech. rep., IRISA report 246, IRISA, Rennes, France (1985).
- [5] D. Harel, Statecharts : A visual approach to complex systems, *Science of Computer Programming* 8 (1987) 231–275.
- [6] D. Harel, A. Pnueli, On the development of reactive systems, in: *Logic and Models of Concurrent Systems*, NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, Vol. 13, NATO ASI series F, Springer Verlag, 1988.
- [7] A. Benveniste, G. Berry, Another look at real-time programming, *Special Section of the Proceedings of the IEEE* 79 (9).
- [8] N. Halbwachs, *Synchronous programming of reactive systems*, Kluwer Academic Pub., 1993.
- [9] X. Nicollin, J. Richier, J. Sifakis, J. Voiron, ATP: an algebra for timed processes, in: *IFIP Working Conference on Programming Concepts and Methods*, Sea of Gallilee, Israel, 1990.
- [10] V. A. Saraswat, *Concurrent Constraint Programming*, ACM Doctoral Dissertation Awards: Logic Programming, The MIT Press, Cambridge, MA, 1993.
- [11] F. Jahanian, A. Mok, Modechart, a specification language for real time systems, *IEEE Transactions on Software Engineering* .
- [12] C. Puchol, A. K. Mok, D. A. Stuart, Compiling Modechart specifications, Technical Report CS-TR-95-38, University of Texas, Austin (Oct. 1, 1995).
URL <ftp://ftp.cs.utexas.edu/pub/techreports/tr95-38.ps.Z>
- [13] A. C. Shaw, Reasoning about time in higher-level language software, *IEEE Transactions on Software Engineering* 15 (7) (1989) 875–889.
- [14] A. Girault, P. Caspi, An algorithm for distributing a finite transition system on a shared/distributed memory system, in: *PARLE'92*, Paris, 1992.

- [15] P. Caspi, C. Mazuet, R. Salem, D. Weber, Formal design of distributed control systems with lustre, in: Proc. Safecom'99, 1999.
- [16] G. Berry, The foundations of estereel, Proof, Language and Interaction: Essays in Honour of Robin Milner Editors: G. Plotkin, C. Stirling and M. Tofte.
- [17] N. Halbwachs, F. Lagnier, C. Ratel, Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE, IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems .
- [18] P. Raymond, D. Weber, X. Nicollin, N. Halbwachs, Automatic testing of reactive systems, in: 19th IEEE Real-Time Systems Symposium, Madrid, Spain, 1998.
- [19] F. Maraninchi, The argos language: Graphical representation of automata and description of reactive systems, in: IEEE Workshop on Visual Languages, Kobe, Japan, 1991.
- [20] F. Maraninchi, Operational and compositional semantics of synchronous automaton compositions, in: CONCUR, LNCS 630, Springer Verlag, 1992.
- [21] D. Harel, On visual formalisms, CACM 31.
- [22] Beeck Michael von der , A Comparison of Statecharts Variants, in: Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), Vol. 863 of Lecture Notes in Computer Science, Springer-Verlag, Lübeck, Germany, 1994, pp. 128–148.
- [23] C. Huizing, Semantics of reactive systems: comparison and full abstraction, Phd thesis, Eindhoven University (1991).
- [24] D. Park, Concurrency and automata on infinite sequences, in: Theoretical Computer Science, 5th GI-Conf., LNCS 104, Springer-Verlag, Karlsruhe, 1981, pp. 167–183.
- [25] R. Milner, Communication and concurrency, in: International Series in Computer Science, Prentice Hall, 1989.
- [26] R. Milner, A calculus of communication systems, in: LNCS 92, Springer Verlag, 1980.
- [27] G. Berry, The constructive semantics of Esterel. Draft book, current version 3.0, July 2, 1999, 2000.
URL
<ftp://ftp.esterel.org/esterel/pub/papers/constructiveness3.ps>
- [28] M. Jourdan, F. Maraninchi, A. Olivero, Verifying quantitative real-time properties of synchronous programs, in: International Conference on Computer-Aided Verification, LNCS 697, Elounda, 1993.
- [29] R. Alur, D. Dill, Automata for modeling real-time systems, in: M. Paterson (Ed.), Proceedings of ICALP 90, Springer Verlag, 1990, pp. 322–335.

- [30] C. Daws, A. Olivero, S. Tripakis, S. Yovine, The tool KRONOS, in: Hybrid Systems III, Verification and Control, Vol. 1066 of LNCS, Springer Verlag, 1996, pp. 208–219.
- [31] D. Simon, K. Kapellos, B. Espiau, M. Jourdan, Formal verification of robotic missions and tasks, in: European Workshop on Hybrid Systems, Grenoble, 1995.
- [32] M. Jourdan, Integrating formal verification methods of quantitative real-time properties into a development environment for robot controllers, in: AMAST Workshop on Real-time Systems, 1996.
- [33] F. Maraninchi, N. Halbwachs, Compositional semantics of non-deterministic synchronous languages, in: European Symposium On Programming, Springer verlag, LNCS 1058, Linkoping (Sweden), 1996.
- [34] N. Halbwachs, F. Maraninchi, On the symbolic analysis of combinational loops in circuits and synchronous programs, in: EUROMICRO, Como, Italy, 1995.
- [35] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers C-35 (8).
- [36] F. Maraninchi, M. Vachon, An experience in compiling a mixed imperative/declarative language for reactive systems, in: International Workshop on Compiler Construction (poster session), Springer Verlag, LNCS 641, 1992.
- [37] J. Fernandez, An implementation of an efficient algorithm for bisimulation equivalence, Science of Computer Programming 13 (2-3).
- [38] A. Arnold, D. Bégay, P. Crubillé, Construction and analysis of transition systems with MEC, World Scientific Publishing, 1994.
- [39] N. Halbwachs, Y. Proy, P. Roumanoff, Verification of real-time systems using linear relation analysis, Formal Methods in System Design 11 (2) (1997) 157–185.
- [40] N. Halbwachs, F. Maraninchi, Y. E. Proy, The railroad crossing problem, modeling with hybrid argos - analysis with polka, in: Second European Workshop on Real-Time and Hybrid Systems, Grenoble (France), 1995.
- [41] C2A-SYNCHRON, The common format of synchronous languages – The declarative code DC version 1.0, Technical report, SYNCHRON project (Oct. 1995).
- [42] F. Maraninchi, N. Halbwachs, Compiling ARGOS into boolean equations, in: Formal Techniques for Real-Time and Fault Tolerance (FTRTFT), Springer verlag, LNCS 1135, Uppsala (Sweden), 1996.
- [43] M. Jourdan, F. Maraninchi, A modular state/transition approach for programming real-time systems, in: ACM Sigplan Workshop on Language, compiler and tool support for real-time systems, ACM Sigplan Notices, Orlando, FL, 1994.

- [44] A. Pnueli, M. Shalev, What is in a step: On the semantics of statecharts, in: Symp. on Theoretical Aspects of Computer Software (STACS), Springer Verlag LNCS 526, 1991.
- [45] D. Harel, A. Pnueli, J. Schmidt, R. Sherman, On the formal semantics of statecharts, in: Symposium on Logic in Computer Science (LICS), 1986, pp. 54–64.
- [46] A. Maggiolo-Schettini, S. Tini, Projectable semantics for statecharts, in: Proceedings of the MFCS Workshop on Concurrency, Electronic Notes in Theoretical Computer Science 18, 1998.
- [47] D. Harel, A. Naamad, The STATEMATE semantics of statecharts, ACM Transactions on Software Engineering and Methodology 5 (4) (1996) 293–333.
URL
<http://www.acm.org/pubs/articles/journals/tosem/1996-5-4/p293-harel/p293-harel.pdf>
- [48] C. Huizing, R. Gerth, W. P. deRoever, Modeling statecharts behavior in a fully abstract way, in: Proceedings on the Colloquium on Trees in Algebra and Programming, Vol. 299 of Lecture Notes in Computer Science, Springer-Verlag, 1988, pp. 271–294.
- [49] A. Uselton, S. Smolka, A process algebraic semantics for statecharts via state refinement, in: IFIP Working Conference on Programming Concepts, Methods and Calculi, Elsevier Science Publishers, San Miniato, Italy, 1994.
- [50] A. C. Uselton, S. A. Smolka, A compositional semantics for statecharts using labeled transition systems, in: B. Jonsson, J. Parrow (Eds.), CONCUR '94: Concurrency Theory, 5th International Conference, Vol. 836 of Lecture Notes in Computer Science, Springer-Verlag, Uppsala, Sweden, 1994, pp. 2–17.
- [51] A. Maggiolo-Schettini, A. Peron, S. Tini, Equivalences of statecharts, in: U. Montanari, V. Sassone (Eds.), CONCUR '96: Concurrency Theory, 7th International Conference, Vol. 1119 of Lecture Notes in Computer Science, Springer-Verlag, Pisa, Italy, 1996, pp. 687–702.
- [52] T. Shiple, G. Berry, H. Touati, Constructive analysis of cyclic circuits, in: International Design and Testing Conference IDTC'96, Paris, France, 1996.
- [53] A. Pnueli, M. Shalev, What is in a step, Tech. rep., Dept. of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Israel (May 1988).
- [54] M. Jourdan, F. Maraninchi, Studying synchronous communication mechanisms by abstractions, in: IFIP Working Conference on Programming Concepts, Methods and Calculi, Elsevier Science Publishers, San Miniato, Italy, 1994.
- [55] M. Jourdan, F. Lagnier, F. Maraninchi, P. Raymond, A multiparadigm language for reactive systems, in: In 5th IEEE International Conference on Computer Languages, IEEE Computer Society Press, Toulouse, 1994.

- [56] C. André, Representation and analysis of reactive behaviors: a synchronous approach, in: IEEE-SMC'96, Computational Engineering in Systems Applications, Lille, France, 1996.
- [57] M. Blanchard, Comprendre, Maitriser et Appliquer le Grafset, Cepadues Editions, 1979.
- [58] F. Maraninchi, Y. Rémond, Mode-automata: About modes and states for reactive systems, in: European Symposium On Programming, Springer Verlag, LNCS 1381, Lisbon (Portugal), 1998.

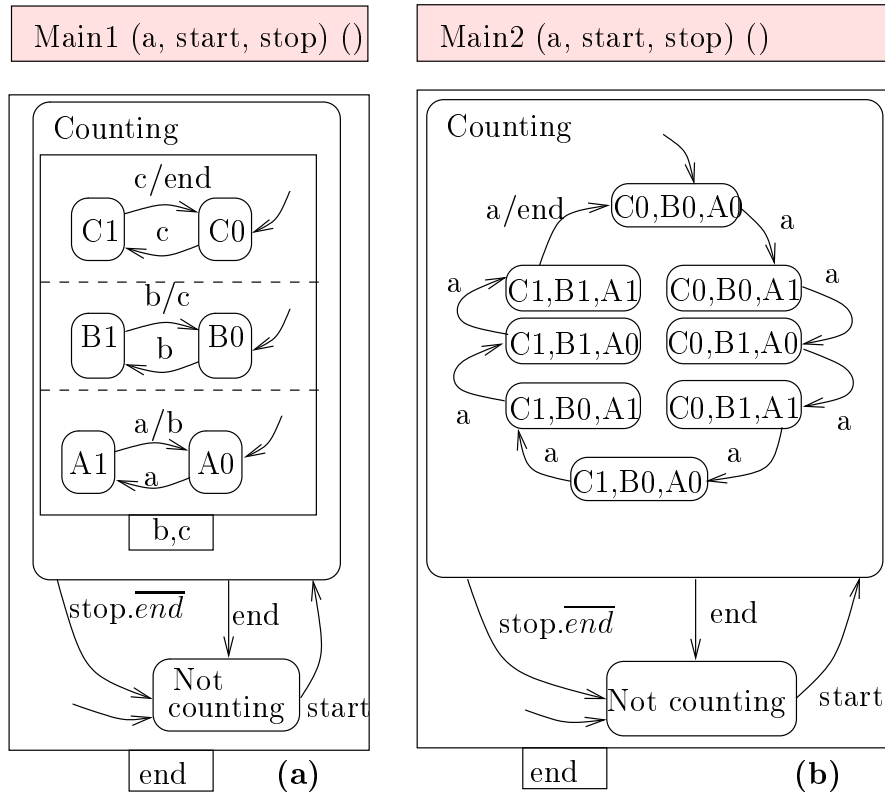


Fig. 1. Two equivalent Argos programs for the modulo-8 a-counter

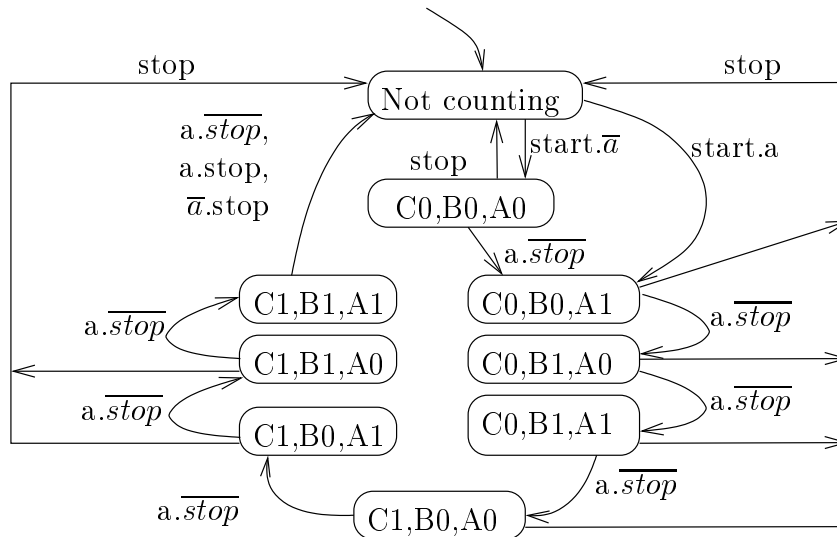


Fig. 2. The behavior of the modulo-8 a-counter

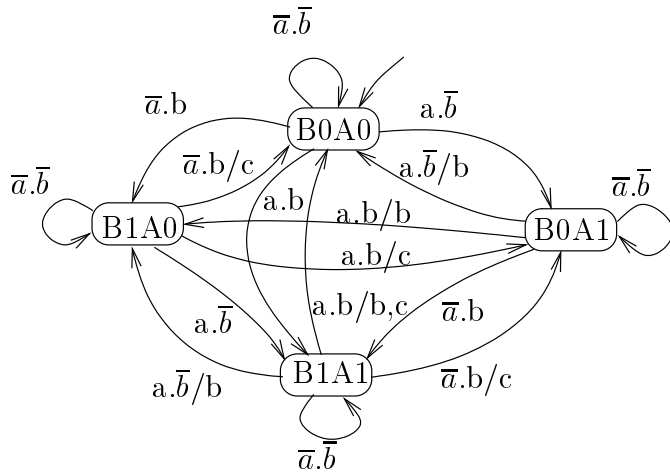
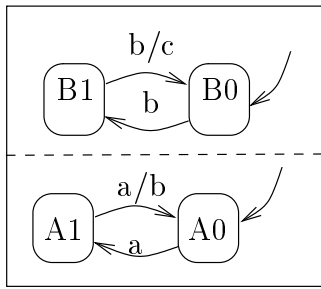


Fig. 3. Semantics of the parallel composition

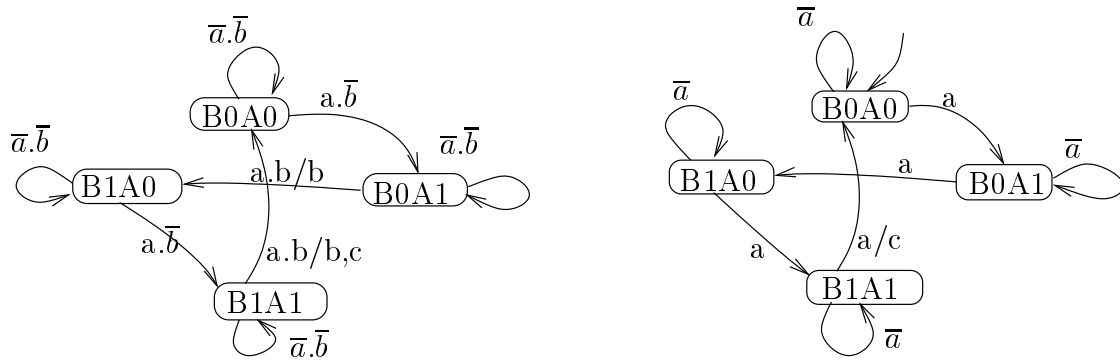


Fig. 4. Semantics of the encapsulation operation: we first apply the criterion of the encapsulation operation to the system obtained from the parallel composition in figure 3, and then we hide the local signal b .

```

module instantaneous_dialogue:

input I;
output S;

signal Q, Y in
[
  await tick;
  present I then emit Q end;
  present Y then emit S end
||
  every Q do emit Y end
]
end.

```

Fig. 5. The instantaneous dialogue in Esterel

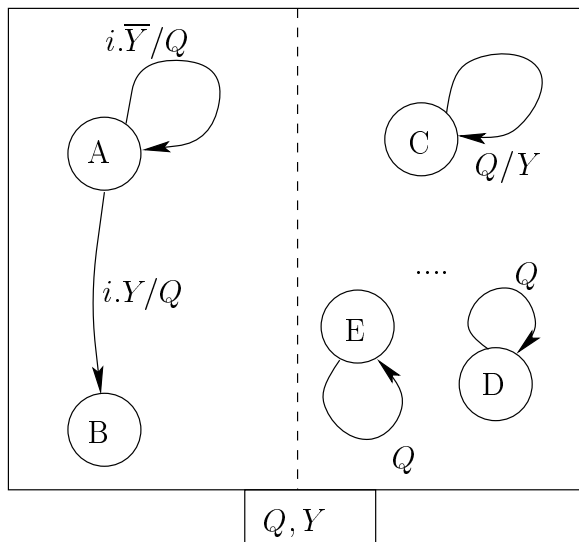


Fig. 6. The instantaneous dialogue with a parallel composition

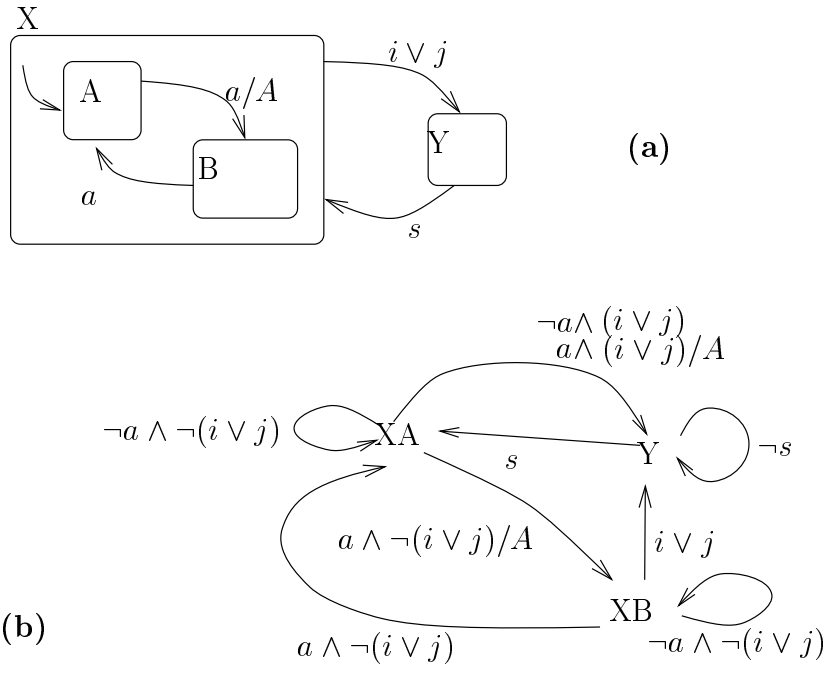


Fig. 7. Refinement: a simple example

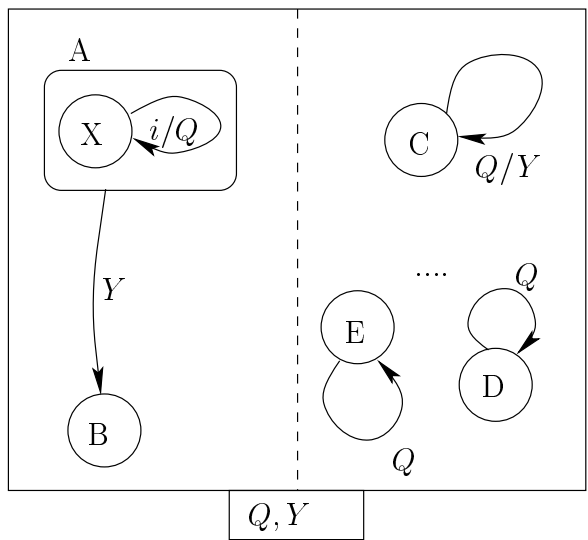


Fig. 8. Instantaneous dialogue, described with a refinement

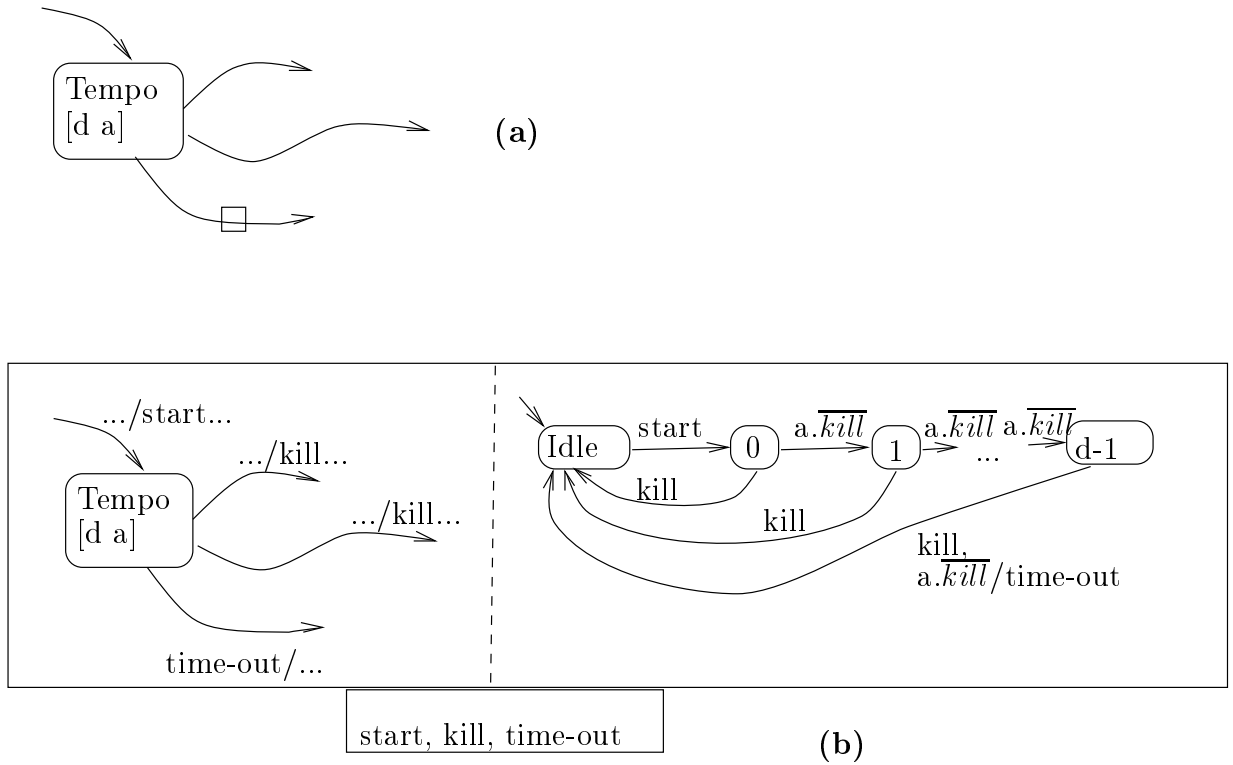


Fig. 9. Macro-notation for temporized states

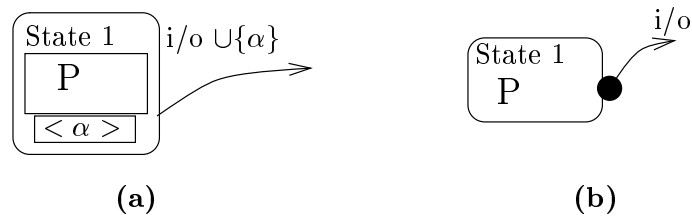


Fig. 10. The macro-notation for inhibiting transitions. The box with $\langle \alpha \rangle$ is the concrete syntax of the inhibiting operator.

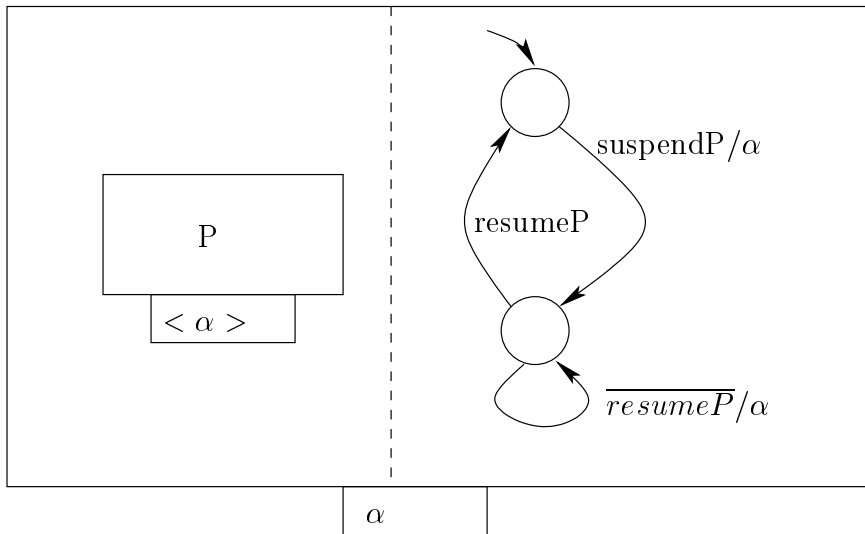


Fig. 11. Suspension of P

Summary