Control machines: A new model of parallelism for compositional specifications and their effective compilation

A.G. Heibig*

LITP, Université PARIS 7, 2 Place Jussieu, 75251 PARIS Cedex 05, France

Communicated by M. Nivat
Received November 1988
Revised May 1989

Abstract


We present a model of parallelism and a specification language which allow modular (compositional) specifications of parallel processes. The specifications specify the states of the processes and broadcasted signals. We also give an algorithm which compiles the main part of this specification language and requires only linear time and space. This specification language is therefore a logical programming language for parallel processes, efficiently compilable.

The model introduces a new communication mean: the synchronization of processes on common states, which implement rigorously instantaneous communications. This is made possible by the introduction of a control environment. The model exhibits a strong connection between temporal logic operators and regular operations on processes. It also allows a natural representation of hierarchical networks of processes and of dynamic creation of processes. We also give a fully abstract semantics based on modular specifications.

The model and the specification language have been designed for real-time applications although the basic ideas are more general.

1. Introduction

Proofs of concurrent processes have been intensely investigated in the last few years. Processes are often specified and proved either in terms of automata [4, 33, 44] or in terms of temporal logic formulas [27, 44, 40]. Even automatic temporal logic checkers, such as CESAR [37], EMC [12] and MEC [1] actually run. Neverthe-

* Present address: LERSI B31, UDESAM, Avenue Normandie Niemen, 13397 Marseille Cedex 13, France.

0304-3975/91/$03.50 © 1991—Elsevier Science Publishers B.V.
less all those proof systems have exponential complexity for concurrent systems, and often even when dealing with a single automaton and a single formula [10, 27]. In fact, Sistla and Clarke [39] proved that for even the simplest linear temporal logics, the model checking problem (checking that a given automaton is a model for a given formula) is PSPACE-complete. On the other hand, the Clarke and Emerson model checker EMC dealing with branching time temporal logic [15], has a linear complexity in both the size of the formula and the size of the automaton. Emerson and Lei [18] also gave small polynomial time model checkers for fragments of the propositional Mu-calculus and efficient model checkers for reasoning under fairness constraints in branching time temporal logic [16, 17]. However because of the non-compositionality of their specifications, all those efficient model checkers have exponential complexity for concurrent systems: in order to model a process $p_1 \parallel p_2$, one has to construct the product automaton of $p_1$ and $p_2$, which size is $|p_1| \cdot |p_2|$. ($p_1$ and $p_2$ denote processes and $p_1 \parallel p_2$ their parallel composition).

Since the verification of concurrent systems is difficult, another approach has been suggested: the direct synthesis of concurrent systems from given specifications. Various methodologies have been proposed: [11, 14, 34, 28, 19, 2, 3, 43]. But again, all those decision procedures have high and often exponential complexity and theoretical results from [26, 39] show that in fact this problem is PSPACE-complete for linear and branching time temporal logic.

This is why compositional (modular) proof systems have also been widely investigated. Roughly speaking, a proof system is modular if it allows the proof of process $p_1 \parallel p_2$ just from specifications of $p_1$ and $p_2$; a special case being when $f(p_1) \land g(p_2) \Rightarrow f \land g(p_1 \parallel p_2)$ holds. ($f$ and $g$ will denote specifications of processes expressed in a temporal logic; $f(p)$ denotes that $p$ satisfies $f$). Such methodologies greatly decrease the verification and synthesis problems, as shown later. The first result was given by Misra and Chandy, [32], whose work has had a great influence on numerous others and especially ours. Let us also mention [38, 42, 36, 47]. The main idea is to reason on the process histories and not on their internal states or on their operations. Although the previous systems are really compositional, it is not unfair to say that they are often complex and very restricted. In fact, there is still a gap between temporal logic, an elegant but nonmodular specification language, and those modular proof systems and specification languages. It has been partially filled by Nguyen et al. [36]. But their proof system does not allow to mention internal states of processes in the modular specifications and is restrictive even when dealing with communications. It also gives no insight on how to synthesize a given specification.

Similar ideas on the histories have been applied in the field of concurrent process semantics and have led to compositional semantics [25, 9, 23, 41]. Let us also mention [31] about compositional semantics. These compositional semantics are intended to provide a basis for compositional specifications and proofs. Such a goal has been aimed for in [20]. But again this proof system seems irksome and difficult to use, mainly because the concurrent systems designer has to deal with very low level details. There is a need for a higher level specification language but we believe
that there is no obvious and easy relationship between such low level semantics and higher level modular specification languages.

Hence we focus on the following aims:

(1) Finding another model of parallelism, for synthesis. The idea is such that the previous complexity results hold only for classical parallelism models.

(2) Finding the modular part of the temporal logic (in this new model), for proofs.

Another aim of this paper, closely related to modularity, is to provide a formal framework for the still informal methodology used by real-time and parallel process designers. Roughly speaking, when designing such a process, one thinks of it as a "black box" having a certain behavior under certain hypotheses. This means that, for practical purposes, this black box has a semantics expressed by a formula ("having a certain behavior under certain hypotheses") rather than by (a set of) traces or a domain of computations. It also means that this semantics (the process specification) is intended to be modular: in any environment satisfying the hypothesis, the process will have the expected behavior.

So in the following, we will only consider modular (compositional or stable by parallel composition) specifications, nonmodular specifications being useless for the programming methodology previously described. A property $f$ is said to be modular iff $\forall p_1, p_2$ processes, $f(p_1) \Rightarrow f(p_1 \parallel p_2)$. Consider for example the process $p$ (Fig. 1). A black point in a state means that there is an initialization control in this state (at $t = 0$ or before). $p$ behaves as follows: if the signal $e$ is present at $t = 0$ (In Presence $e$), $p$ executes the transition toward $s_2$. Else, it executes the transition toward $s_1$. Then $\text{EX}s_1$ holds, meaning that there exists (E) an execution of $p$ such that at time $t = 1$ (X), $p$ is in $s_1$.

Suppose now that $p$ works in parallel with a process $p_1$ which forces $\text{ip} e$ for $p$ ($p_1$ emits $e$ at time $t = -1$). Then $\text{AX}s_2(p \parallel p_1)$ holds, meaning that for every (A) execution of $p \parallel p_1$, at time $t = 1$ (X), $p \parallel p_1$ is in $s_2$. Then the property $\text{EX}s_1(p)$ is useless if we consider $p \parallel p_1$. That is to say that considering nonmodular properties oblige to consider processes working alone. In particular this forbids parallel or reactive processes and program development.

On the other hand, modular specifications allow a natural programming methodology:

(1) Synthesis: If we are trying to realize $f = g \land h$ (trying to find a model $p$ for $f = g \land h$), we just have to find $p_1$ and $p_2$ realizing $g$ and $h$ (or $g \Rightarrow h$ and $g$). Then we have split the original problem into two smaller ones.

![Fig. 1.](image-url)
(2) **Proof:** Conversely, if we are trying to prove that an already written program $p = p_1 \| p_2$ realizes (is a model for) $f = g \land h$, we just have to prove that $p_1$ and $p_2$ realize $g$ and $h$.

(3) **Compositional semantics:** $g$ and $h$ may be thought of as $p_1$ and $p_2$ semantics. Then the $p = p_1 \| p_2$ semantics is $f = g \land h$.

(4) **Debugging:** Suppose that the program $p = p_1 \| p_2$, which should realize $g \land h$ by means of $g(p_1)$ and $h(p_2)$, is false because $g(p_1)$ does not hold. In order to correct it, we just have to correct $p_1$; there is no need to modify $p_2$.

(5) **Program development:** If we now want to realize $f \land g \land h$, we must have to find $p_3$ such that $h(p_3)$; then $f \land g \land h(p_1 \| p_2 \| p_3)$ holds. Again there is no need to modify $p_2$ (and $p_1$).

Notice that sometimes it is even possible to realize compositionally a nonmodular formula. Let $Af$ be a nonmodular formula, $Ag$ a modular one such that $Ag \Rightarrow Af$, and $CM$ a control machine realizing $Ag$: $CM Ag$. Then $\forall CM', CM \parallel CM' Ag$ and then $CM \parallel CM' Af$. So $CM$ compositionally realizes $Af$, even if $Af$ is not compositional.

In conclusion, we will only consider modular specifications in the following. We will also show that modular specifications allow to define a fully abstract semantics.

Section 2 presents general ideas about modular specifications. Sections 3 and 4 introduce a new model of parallelism and Section 5 introduces the operational semantics of processes. Section 6 and 7 present the specification language. In Section 8, we introduce a modular sub-language of the specification language. Then, in Section 9, we prove that under certain (natural) hypotheses, our model is the only one which achieves modularity. Sections 10 and 11 present extensions of the modular specification sub-language. Section 11 extension leads to a semantics fully abstract with respect to an observational one. Section 12 shows that the previous modularity results may be extended to environments. Section 13 gives a synthesis algorithm of linear complexity. Section 14 applies the previous ideas to hierarchical design of processes.

### 2. Modularity

Extending Misra's and Chandy's ideas [32], we will define stable specifications of process executions. Let $p_1, p_2, p_3$ be processes. Let $\text{exec}(p_1 ; p_2 \| p_3)$ be an execution of $p_1$ in the environment $p_2 \| p_3$ and let $\text{exec}(p_1 \| p_2 ; p_3)$, execution of $p_1 \| p_2$ in the environment $p_3$, be the same execution seen from the $p_1 \| p_2$ point of view (i.e. $\text{exec}(p_1 ; p_2 \| p_3)$ and $\text{exec}(p_1 \| p_2 ; p_3)$ are making the same nondeterministic choices). Let $f$ be an execution specification. We define

\[
[f \text{ stable}] \Leftrightarrow [\forall p_1 ; p_2 ; p_3 ; \forall \text{exec}(p_1 ; p_2 \| p_3);
\]

\[f \text{ exec}(p_1 ; p_2 \| p_3) \Rightarrow f \text{ exec}(p_1 \| p_2 ; p_3)].\]
Now suppose that $p_1$ satisfies $Af(p_1)$ with (definition)

$$[Af(p_1)] \iff [\forall p_2 ; \forall \text{exec}(p_1 ; p_2) ; f \text{exec}(p_1 ; p_2)]$$

Then, if $f$ is stable

$$[Af(p_1)] \Rightarrow [\forall p_2 ; p_3 ; \forall \text{exec}(p_1 ; p_2 \parallel p_3) ; f \text{exec}(p_1 ; p_2 \parallel p_3)]$$

$$\Rightarrow [\forall p_2 ; p_3 ; \forall \text{exec}(p_1 \parallel p_2 ; p_3) ; f \text{exec}(p_1 \parallel p_2 ; p_3)]$$

$$\Rightarrow [Af(p_1 \parallel p_2)].$$

Then $Af(p_1) \Rightarrow Af(p_1 \parallel p_2)$ holds, meaning that $Af$ is a modular property: So $Af$ is a modular property if $f$ is stable. The two fundamental points are (1) to use a specification quantified by the universal quantifier (2) where $f$ is stable. ($f$ is stable just means that if $f$ is true from $p_1$ point of view, it is also true from the $p_1 \parallel p_2$ one. $f$ could be called invariant too).

This result is valid for any specification language, a temporal logic or not; it is also valid for any model of parallelism. In particular, the processes may communicate by synchronous or asynchronous communications, by broadcasting or buffer or rendez-vous . . .

We will now give an example to formalize the previous ideas in the case where the specifications belong to a temporal logic and the processes have a particular form.

3. The model of parallelism

The elementary processes are extensions of finite input–output automata called control machines (CM). Time is discontinuous. At time $t$, a control machine is in one or more states $s_j(t)$ (has one or more controls). Depending on its environment, it will execute several transitions, being in several states. The transitions are nondeterministic. They bring it at $t+1$ in state $s_k(t+1)$. A control machine is initialized at $t_{CM} \leq 0$ with $S_1$ its initial states set: at $t_{CM}$, this machine creates controls in all the states of $S_1$. A control machine may emit signals. They are stocked in an environment called signal environment (EnvE') [6]. If a control machine emits the signal $e$ at time $t$, then $e$ will be in EnvE' at $t+1$. EnvE' is erased between two instants: $e \in \text{EnvE'}(t+1)$ does not allow to deduce that $e \in \text{EnvE'}(t+2)$. Any control machine may read or write in it.

A process $p$ is a set of control machines working in parallel, that is to say executing their transitions simultaneously, as if they were alone, but reading and writing signals in the same signal environment. (Notice that we will often call a control machine a process, i.e. a set of control machines). Between $t$ and $t+1$ every control machine of $p$ had made at least a transition (there is no blocking state). We have just seen that if the control machines CM$_1$ and CM$_2$ are working in parallel, their signals are stocked in the same signal environment. We do the same for the states (controls): we stock the controls of two different control machines in a same control
environment EnvC'. (EnvC, EnvC', EnvE, EnvE' are functions of time). This must be understood as follows: if s is a state common to processes p1 and p2 and if p1 is in s at time t, then we force p2 to be in s at t. So a process p2 may be in s at t for two very different reasons:

1. Because p2 had executed itself at t - 1 a transition toward s. This is the classical case.
2. Because p2 is working in parallel with p1 which arrives in s at t, s being common to p1 and p2.

The second mechanism, called "state identification", is in fact an instantaneous communication (and synchronization) mechanism between processes.

The processes are working in environments constituted by processes or other "actors" which are not or which cannot be modelled by processes (the operator for example). An environment creates a control environment (EnvC) and a signal environment (EnvE). So the process p2 may reach s at t for another reason:

3. Because s ∈ EnvC at t.

The second and third mechanisms can be easily implemented. A process is implemented as a table of transitions describing each of its transitions. A control machine is implemented as a single transition table compounded of the transition tables of its subprocesses. A set of control machines CMi working in parallel is also implemented as a single transition table, compounded of the transitions tables of each of the CMi. (Notice that the transition table is conceptually unique but can be implemented on several computers). The controls and signals are implemented by common environments, as described previously. If at t, p1 arrives in s (mechanism 2), then s is written in EnvC'(t). Then between t and t + 1, p1 will execute a transition from s. But p2 will also execute (another) transition from s. Between t and t + 1, the transition table, which includes the transitions of p2 (and also those of p1), will be scanned and, as s belongs to EnvC'(t), one or more transitions of p2 from s will be activated. The third and the first mechanisms work in a similar way.

EnvC and EnvE are supposed to be fixed and independent of p1 and p2 reactions. The process p1 || p2, working in the environment (EnvC; EnvE) creates himself new signals and controls. We put them together with a copy of (EnvC; EnvE) and get (EnvC'; EnvE'). So (EnvC'; EnvE') contain all the states and signals while (EnvC; EnvE) describe the environment independent of p1 and p2 executions. Physically, EnvC and EnvC' are in two different places, even if EnvC' contains a copy of EnvC. Notice that EnvC and EnvE may be the actions of an environment of others processes, stocked somewhere else than in EnvC' and EnvE'. For example let p3 be a member of this environment; p3 will write its controls and signals in (EnvC; EnvE) and in (EnvC'; EnvE'). So p1 and p2 will read p3, controls and signals. But conversely, p3 will read its controls and signals not in (EnvC'; EnvE') but in (EnvC; EnvE). So p1 and p2 will not influence p3 and (EnvC; EnvE). This leads to a hierarchy of levels of processes, where p3 is at a higher level than p1 and p2, which are at the same level. So p3 may influence p1 and p2 but the converse is false. p1 and p2 may influence one another.
The transitions have the form described in Fig. 2. This means that if

\[ s_1; s'_1 \ldots \in \text{EnvC}'(t), \quad s_2; s'_2 \ldots \notin \text{EnvC}'(t), \]

\[ e_1; e'_1 \ldots \in \text{EnvE}'(t), \quad e_2; e'_2 \ldots \notin \text{EnvE}'(t), \]

then the process will execute this transition emitting \( e_3; e'_3 \ldots \). Notice that the preconditions of the transition name both the states and the signals while the actions realized by this transition only use signals (but arrives in states \( s_1 \) at \( t+1 \)). This avoids temporal paradoxes such as the one described on Fig. 3 where the state \( s_2 \) would be broadcasted immediately, so \( \text{ia} s_2 \Rightarrow \text{ia} s_2 \) would hold.

We will also allow transitions such as those of Fig. 4 meaning that in presence of \( e \), the control machine simultaneously executes the two transitions. This will also be denoted as in Fig. 5. In the same way, we will denote, as in Fig. 6, when the process may choose nondeterministically between the transition toward \( s_1 \) or the one toward \( s_2 \). The general transition form is the one depicted in Fig. 7.
4. The formal definition of a control machine

Let $S$ be a set of states, $E$ a set of signals and $P(S)$ the power set of $S$. Consider the functions of time $\text{Env}C$ ($\text{Env}C'$) and $\text{Env}E$ ($\text{Env}E'$),

$$\text{Env}C : \mathbb{Z} \rightarrow P(S) \quad \text{Env}E : \mathbb{Z} \rightarrow P(E)$$

$t \rightarrow \text{Env}C(t) \quad t \rightarrow \text{Env}E(t)$.

$t$ ranges over $\mathbb{Z}$ for notational convenience. $\text{Env}C$ ($\text{Env}E$) is definitely fixed. $\text{Env}C'$ ($\text{Env}E'$) will be defined later. A control machine $CM$ is an $n$-tuple: $CM = (X; Y; S(CM); t_{CM}; S_I; \delta)$. Notice that there is no set of terminal states as in the classical definition of an input-output automaton. This is so because our processes are supposed to be working continuously with no blocking state.

- $S(CM) \subseteq S$ is the set of CM states.
- $CM$ is supposed to be activated at $t_{CM} \leq 0$ from its set of initial states $S_I$ ($S_I \subseteq S(CM)$). This means that at $t_{CM}$, $CM$ will add $S_I$ to $\text{Env}C'$.
- $X$ is the set of inputs: $X \subseteq P(S) \times P(S) \times P(E) \times P(E)$. An input $x = (S_1; S_2; E_1; E_2) \in X$ is such that $S_1 \cap S_2 = \emptyset$ and $E_1 \cap E_2 = \emptyset$. $S_1$ and $S_2$ ($E_1$ and $E_2$) are the sets of states (signals) which must be present and absent. $x$ will also be noted $(\text{ip} S_1; \text{ia} S_2; \text{ip} E_1; \text{ia} E_2)$.
- $Y$ is the set of outputs: $Y \subseteq P(E)$.
- $\delta$ is the transition function. $\delta : S(CM) \times X \rightarrow P(Y \times S(CM))$. We can also note $\delta \subseteq S(CM) \times X \times P(Y \times S(CM))$.

Let

$$\delta(s_0, (\text{ip} S_1, \text{ia} S_2, \text{ip} E_1, \text{ia} E_2))$$

$$= \{\{(e_{1,1,1}; \ldots; e_{1,1,\beta_1}; s_{1,1}) \ldots \{(e_{1,\alpha_1,1}; \ldots; e_{1,\alpha_1,\beta_1}; s_{1,\alpha_1})\}\}$$

$$\ldots$$

$$\{\{(e_{k,1,1}; \ldots; e_{k,1,\beta_k}; s_{k,1}) \ldots \{(e_{k,\alpha_k,1}; \ldots; e_{k,\alpha_k,\beta_k}; s_{k,\alpha_k})\}\}$$

$$\ldots$$

$$\{\{(e_{m,1,1}; \ldots; e_{m,1,\beta_m}; s_{m,1}) \ldots \{(e_{m,\alpha_m,1}; \ldots; e_{m,\alpha_m,\beta_m}; s_{m,\alpha_m})\}\}.$$
This means that if CM is in \( s_0 \) at \( t \) and if the states of \( S_1 \) are present in \( \text{EnvS} \) (\( S_1 \subseteq \text{EnvS} \)) and those of \( S_2 \) are absent in \( \text{EnvS} \) (\( S_2 \cap \text{EnvS} = \emptyset \)), and so for \( E_1 \) and \( E_2 \), then nondeterministically, CM will choose one (or more) \( k \) and will execute the transitions toward \( s_{k,1}, \ldots, s_{k,\ell_k} \) emitting for these transitions \( \{e_{k,1,1}; \ldots; e_{k,1,\beta_k}\} \ldots \{e_{k,\ell_k,1}; \ldots; e_{k,\ell_k,\beta_k}\} \). \( \delta \) must be such that there is no blocking state.

5. Process executions

Let \( P_1 \) and \( P_2 \) be two deterministic processes (the nondeterministic case is similar). We now write \( C, C', E, E' \) for \( \text{EnvC}, \text{EnvC'}, \text{EnvE}, \text{EnvE'} \). We denote by \( \text{exec}(p_1, p_2, C, E) \) the execution of \( p_1 \) (and not of \( p_1 || p_2 \)) while working in parallel with \( p_2 \) in the environment \((C, E)\). (In the nondeterministic case we should consider \( \text{Exec}(p_1, p_2, C, E) \), the corresponding execution set.) \( \text{exec}(p_1, p_2, C, E) \), also written as exec, is a function of time

\[
\text{exec}: \mathbb{Z} \to P(S) \times P(E) \times P(S) \times P(E), \quad t \to (\text{exec}_1(t), \text{exec}_2(t), \text{exec}_3(t), \text{exec}_4(t)).
\]

Intuitively, \( \text{exec}_1(t) \) is the set of \( p_1 \) states (and not those of \( p_1 \parallel p_2 \)) at \( t \) for the execution \( \text{exec} \); \( \text{exec}_2(t) \) is the set of signals emitted by \( p_1 \) at \( t \); \( \text{exec}_3 = C'(t) \); \( \text{exec}_4 = E'(t) \).

\( C'(t) \) is still the set of all the present states, of \( p_1, p_2 \) and \( C \). In the same way, \( E'(t) \) is the set of all the signals present at \( t \): those emitted by \( p_1 \) or \( p_2 \) at \( t-1 \) and those belonging to \( E(t) \). Then we have the following definitions:

\[
\begin{align*}
\text{s}_0(t) & \iff s_0 \in \text{exec}_1(t), & \text{em} e_0(t) & \iff e_0 \in \text{exec}_2(t), \\
\text{ip} s_0(t) & \iff s_0 \in \text{exec}_3(t), & \text{ip} e(t) & \iff e \in \text{exec}_4(t).
\end{align*}
\]

\( S(p_1) \) is the set of \( p_1 \) states. We then define

\[
\begin{align*}
(1) \quad C'(t) & = C(t) \cup \text{AS}(p_1; t-1) \cup \text{AS}(p_2; t-1) \cup I(p_1; t) \cup I(p_2; t), \\
(2) \quad E'(t) & = E(t) \cup \text{exec}_2(p_1; t-1) \cup \text{exec}_2(p_2; t-1), \\
(3) \quad \text{exec}_1(t) & = C'(t) \cap S(p_1), \\
(4) \quad \text{exec}_2(t) & = \Omega(\text{exec}_1(t); C'(t); E'(t)),
\end{align*}
\]

where \( \text{AS}(p_1, t-1) \) is the set of the states attained at \( t \) by \( p_1 \) by means of a transition (then started in another state of \( p_1 \) at \( t-1 \)).

(1) \( I(p_1; t) \) is the set of controls created in \( p_1 \) at \( t \) by means of an initialization of a control machine belonging to \( p_1 \)

\[
I(p_1; t) = \{ s \in S | \exists p_1; t_{CM} = t \text{ and } s \in S_t(CM) \}.
\]
We should have had \( C'(t) = C(t) \cup S(p_1; t) \cap S(p_2; t) \) where \( S(p_1; t) \) is the set of \( p_1 \) states at \( t \). But
\[
S(p_1; t) = AS(p_1; t-1) \cup [AS(p_2; t-1) \cap S(p_1)] \cup I(p_1; t)
\]
and then
\[
C'(t) = C(t) \cup S(p_1; t) \cup S(p_2; t)
\]
\( = C(t) \cup AS(p_1; t-1) \cup AS(p_2; t-1) \cup I(p_1; t) \cup I(p_2; t), \)
which is (1).

(2) means that there is a delay of one time unit between the emission and the reception of a signal.

(3) means that if \( p_2 \) is in \( s \) at \( t \) and if \( s \) is common to \( p_1 \) and \( p_2 \), then \( p_1 \) is in \( s \) at \( t \) too. Note that this definition allows a process to be in several states at the same time.

(4) means that \( p_1 \) emissions at \( t \) are functions of \( p_1 \) states and of its signal (ip \( e_1 \); ia \( e_1 \)) and control (ip \( s_0 \); ia \( s_0 \)) environments.

We define now \( AS \) and \( \Omega \). Let \( x = (ip S_1; ia S_2; ip E_1; ia E_2) \in X \). Then \( S_j \subseteq S; E_h \subseteq E; S_1 \cap S_2 = \emptyset; E_1 \cap E_2 = \emptyset \). We say that \( x \) accepts \( C'(t) [E'(t)] \) iff
\[
S_1 \subseteq C'(t) \quad S_2 \cap C'(t) = \emptyset \quad [E_1 \subseteq E'(t); E_2 \cap E'(t) = \emptyset].
\]

In the same way let \( y = (em E_3) \in Y; (E_3 \subseteq E) \). We say that \( e \) belongs to \( y \) iff \( e \in E_3 \). Then we define
\[
s_0 \in AS(p_1; t) \iff [\exists s_1 \in \text{exec}_1(p_1; t-1); \exists \text{CM}_1 \text{ control machine of } p_1; \text{CM}_1 = (X; Y; t_{CM_1}; S(\text{CM}_1); S_{11}; \delta_1); \exists d_1 \in \delta_1 \text{ such that } d_1 = (s_1; x_1; y_1; s_0) \text{ and } x \text{ accepts } C'(t-1) \text{ and } E'(t-1)].
\]

In the same we define
\[
e \in \text{exec}_2(p_1; t) \iff [\exists s_1 \in \text{exec}_1(p_1; t); \exists \text{CM}_1 \in p_1; \text{CM}_1 = (X; Y; t_{CM_1}; S_{11}; \delta_1); \exists d_1 \in \delta_1; d_1 = (s_1; x_1; y_1; s_2) \text{ and } C'(t-1) \text{ and } E'(t-1) \text{ are accepted by } x_1 \text{ and } e \text{ belongs to } y_1].
\]

The previous definitions allow to construct \( C'(t); E'(t); \text{exec}_1(t); \text{exec}_2(t) \) for each \( t \).

6. The specification language: MCTL*

Our syntax is nearly that of [15].
We define a set of state formulas (abbreviated as SF) and a set of execution formulas (EF). These execution formulas will not describe paths but “trees”
Control machines: a new model of parallelism

6.1. Syntax

S2: If $f$ and $g$ are state formulas, so are $f \land g$ and $\neg f$.

S3: If $f$ is an execution formula, then $Af$ and $Ef$ are state formulas.

P1: Every atomic proposition is an execution formula. The atomic proposition are $s_0; ip s_0; ip e; em e$.

P2: If $f$ and $g$ are execution formulas, so are $f \land g$ and $\neg f$.

P3a: If $f$ is a state formula, then $Gf$ and $Ff$ are execution formulas.

P3b: If $f$ is an execution formula, then $Gf$ and $Ff$ are execution formulas.

P4a: If $f$ and $g$ are state formulas, then $Xf$ and $f \lor g$ are execution formulas.

P4b: If $f$ and $g$ are execution formulas, then $Xf$ and $f \lor g$ are execution formulas.

We write $ia e$ (in absence $e$) and $ia s_0$ (in absence $s_0$) for $\neg ip e$ and $\neg ip s_0$. We introduce a new operator: $N$ (now).

P5a: If $f$ is a state formula, then $Nf$ is an execution formula.

P5b: If $f$ is an execution formula, so is $Nf$.

6.2. Semantics

Let $(C, E)$ be a fixed environment. We give the following definitions:

S2: $(C, E) \models f \land g$: classical definitions (so for \lor and $\neg$).

S3: $(C, E) \models Af(p_1) \Leftrightarrow \forall p_2; \forall \text{exec}(p_1, p_2, C, E); f \text{exec}]$,

$(C, E) \models Ef(p_1) \Leftrightarrow (C, E) \models \neg A \neg f(p_1)$.

$p_1 \models Af$ will often be denoted by $Af(p_1)$. We have the very important property

$p_1 \models Af \Leftrightarrow [\forall C; \forall E; (C, E) \models Af(p_1)]$,

$p_1 \models Ef \Leftrightarrow \neg [p_1 \models A \neg f]$.

In the following we will study the modularity of the formula $p_1 \models Af$, which is a consequence of this property. We now define $f \text{exec}(p_1, p_2, C, E)$ which gives the semantics of the axioms $P_j$. We first define $(C, E) \models f(p_1, p_2)$ where $f$ is a state formula.

S3: $[(C, E) \models Af(p_1, p_2)] \Leftrightarrow [\forall p_3; \forall \text{exec}(p_1, p_2, p_3, C, E); f \text{exec}]$,

$[(C, E) \models Ef(p_1, p_2)] \Leftrightarrow [(C, E) \models \neg A \neg f(p_1, p_2)]$. 
Then we define

**P1:** \( s_0 \text{ exec} \Leftrightarrow s_0 \in \text{exec}_1(0), \)
\( \text{ip } s_0 \text{ exec} \Leftrightarrow s_0 \in C'(0) = \text{exec}_3(0) \)
\( \text{em } e \text{ exec} \Leftrightarrow e \in \text{exec}_2(0), \)
\( \text{ip } e \text{ exec} \Leftrightarrow e \in E'(0) = \text{exec}_4(0). \)

**P2:** classical definitions.

**P3a:** \( f \) state formula,
\[
[\text{Gf exec}] \Leftrightarrow [\forall t \geq 0; (C_t \cup C'(t); E_t \cup E'(t)) \models f(p_1; p_2)],
\]
\[
[\text{Ff exec}] \Leftrightarrow [\exists t \geq 0; (C_t \cup C'(t); E_t \cup E'(t)) \models f(p_1; p_2)].
\]

\( C_t(E_t) \) is the suffix of \( C(E) \) beginning at \( t \). Notice that, as \( C'(t) \) is only defined at \( t \), the notation \( C_t \cup C'(t) \) is in fact incorrect. Notice too that in \( \forall t \) the quantifier \( \forall \) ranges over \( \mathbb{N} \) and not over \( \mathbb{Z} \).

We have to keep \( p_2 \) in this definition in order to achieve modularity. This will be detailed later.

**P3b:** \( f \) execution formula,
\[
[\text{Gf exec}] \Leftrightarrow [\forall t \geq 0; f \text{exec}_t],
\]
\[
[\text{Ff exec}] \Leftrightarrow [\exists t \geq 0; f \text{exec}_t].
\]

where \( \text{exec}_t \) is the suffix of \( \text{exec} \) beginning at \( t \). One easily proves that: \( \text{exec}_t(t') = \text{exec}(t' + t) \) and
\[
\text{exec}_t(p_1, p_2, C, E) = \text{exec}(p_1, p_2, C_t \cup C'(t), E_t \cup E'(t)).
\]

This equality allows us to consider the \( t < 0 \) as an initialization phase leading to the right environment at \( t = 0 \). In the nondeterministic case we should work on execution sets. Otherwise, if we want to keep the same equalities we must consider the \( \text{exec}(p_1, p_2, C_t \cup C'(t), E_t \cup E'(t)) \) making the same nondeterministic choices as the considered \( \text{exec}_t(p_1, p_2, C, E) \).

**P4a:** \( f \) and \( g \) state formulas,
\[
Xf \text{ exec} \Leftrightarrow (C_1 \cup C'(1); E_1 \cup E'(1)) \models f(p_1, p_2),
\]
\[
f \cup g \text{ exec} \Leftrightarrow \exists t'; \forall 0 \leq t \leq t'; (C_t \cup C'(t); E_t \cup E'(t)) \models f(p_1; p_2)
\]
and \( (C_t \cup C'(t); E_t \cup E'(t)) \models g(p_1; p_2) \].

**P4b:** \( f \) and \( g \) execution formulas,
\[
[Xf \text{ exec}] \Leftrightarrow [f \text{exec}_t],
\]
\[
[f \cup g \text{ exec}] \Leftrightarrow [\exists t'; \forall 0 \leq t \leq t'; f \text{exec}_t \text{ and } g \text{exec}_t].
\]

**P5a:** \( f \) state formula,
\[
Nf \text{ exec} \Leftrightarrow (C_0 \cup C'(0); E_0 \cup E'(0)) \models f(p_1, p_2).
\]
P5b: $f$ execution formula,

\[ Nf = f. \]

The meaning of $\text{ANA}f'$ is the following: consider first $\text{AXA}f'(p_1)$. This means that, for every process $p_2$ working in parallel with $p_1$ and for every execution $(A)$, at the next moment, i.e. with the controls and signals present at $t = 1$ $(X)$, the property $A^f'(p_1; p_2)$ holds. While $\text{AXA}f'$ deals with $t = 1$, $\text{ANA}f''$ deals with $t = 0$. $\text{ANA}f''(p_1)$ means that, for every process $p_2$ working in parallel with $p_1$ and for every execution $(A)$, at that moment, i.e. with the controls and signals present at $t = 0$ $(N)$, the property $A^f'(p_1; p_2)$ holds.

7. Normal form

We assume that every formula $f$ has been placed in normal form, with all negations and all $X$ driven inside so that (1) only atomic propositions can be negated and (2) only atomic propositions or their negations can be after an $X$. We obtain $\{f\}$ by means of the following rewriting system, which trivially terminates: ($\rightarrow$ is in fact an equality).

$f$ state formula:

\[
\begin{align*}
\{\neg f\} & \rightarrow \{f\}, \\
\{\neg(f \land g)\} & \rightarrow \{\neg f\} \lor \{\neg g\}, \\
\{\neg(f \lor g)\} & \rightarrow \{\neg f\} \land \{\neg g\}, \\
\{\neg A f\} & \rightarrow E\{\neg f\}, \\
\{\neg E f\} & \rightarrow A\{\neg f\}, \\
\{f \land g\} & \rightarrow \{f\} \land \{g\}, \\
\{f \lor g\} & \rightarrow \{f\} \lor \{g\}, \\
\{A f\} & \rightarrow A\{f\}, \\
\{E f\} & \rightarrow E\{f\},
\end{align*}
\]

$f$ execution formula:

\[
\begin{align*}
\{p\} & \rightarrow p \quad \text{if } p \text{ is an atomic proposition or its negation} \\
\{\neg f\} & \rightarrow \{f\}, \\
\{\neg G f\} & \rightarrow F\{\neg f\}, \\
\{\neg F f\} & \rightarrow G\{\neg f\}, \\
\{\neg (f_1 U f_2)\} & \rightarrow \{[\neg f_2 \land \{f_1\}] U (\neg f_2 \land \{\neg f_1\})\} \lor G\{\neg f_2\}, \\
\{\neg (f_1 U f_2)\} & \rightarrow \{[\neg f_2 \land \{f_1\}] U (\neg f_2 \land \{\neg f_1\})\}, \\
\{\neg X f\} & \rightarrow X\{\neg f\}, \\
\{f \land g\} & \rightarrow \{f\} \land \{g\}, \\
\{f \lor g\} & \rightarrow \{f\} \lor \{g\}, \\
\{G f\} & \rightarrow G\{f\}, \\
\{F f\} & \rightarrow F\{f\}, \\
\{f_1 U f_2\} & \rightarrow \{f_1\} U \{f_2\}, \\
\{f_1 U f_2\} & \rightarrow \{f_1\} U \{f_2\}, \\
\{f_1 U f_2\} & \rightarrow \{f_1\} U \{f_2\}, \\
\{f_1 U f_2\} & \rightarrow \{f_1\} U \{f_2\}.
\end{align*}
\]
$Xf$: $f$ state formula:

\[
\{X \neg f\} \rightarrow X\{\neg f\},
\]
\[
\{X(f \land g)\} \rightarrow \{Xf\} \land \{Xg\}, \quad \{X(f \lor g)\} \rightarrow \{Xf\} \lor \{Xg\},
\]
\[
\{XAf\} \rightarrow XA\{f\}, \quad \{XEf\} \rightarrow XE\{f\}.
\]

$Xf$: $f$ execution formula:

\[
\{X \neg f\} \rightarrow X\{\neg f\},
\]
\[
\{X(f \land g)\} \rightarrow \{Xf\} \land \{Xg\}, \quad \{X(f \lor g)\} \rightarrow \{Xf\} \lor \{Xg\},
\]
\[
\{XXf\} \rightarrow XX\{f\}, \quad \{XGf\} \rightarrow GX\{f\}, \quad \{XFf\} \rightarrow FX\{f\},
\]
\[
\{X(f_1 \cup f_2)\} \rightarrow \{Xf_1\} \cup \{Xf_2\}, \quad \{X(f_1 \tilde{\cup} f_2)\} \rightarrow \{Xf_1\} \tilde{\cup} \{Xf_2\}.
\]

where we defined

\[
(f_1 \tilde{\cup} f_2) = (f_1 \cup f_2) \lor Gf_1.
\] (3)

Then we have

\[
(f_1 \cup f_2) = (f_1 \tilde{\cup} f_2) \land Ff_2.
\] (4)

The proofs may be found in [22].

We have the following axioms and rules of inference, similar to those given by Emerson and Halpern in [14] for CLT.

**Axioms:**

(Ax 1) All the tautologies of propositional logic.

(Ax 2) $EFp = E(true \lor p)$,

(Ax 3) $AFp = A(true \lor p)$,

(Ax 4) $EX(p \lor q) = EXp \lor EXq$,

(Ax 5) $AXp = \neg EX \neg p$,

(Ax 6) $E(p \lor q) = E[q \lor (p \land X(p \lor q))]$ if $p$ and $q$ EF, (*)

(Ax 7) $A(p \lor q) = A[q \lor (p \land X(p \lor q))]$ if $p$ and $q$ EF, (*)

(Ax 6') $E(p \lor q) = E[Nq \lor (Np \land X(p \lor q))]$ if $p$ and $q$ SF, (*)

(Ax 7') $A(p \lor q) = A[Nq \lor (Np \land X(p \lor q))]$ if $p$ and $q$ SF, (*)

(Ax 8) $EXtrue \land AXtrue$.

**Rules of inference:**

(R1) $p \Rightarrow q \vdash EXp \Rightarrow EXq$,

(R2) $r \Rightarrow (\neg q \land EXr) \vdash r \Rightarrow \neg A(p \lor q)$,

(R3) $r \Rightarrow [\neg q \land AX(r \lor \neg E(p \lor q))] \vdash r \Rightarrow \neg E(p \lor q)$,

(R4) $p; (p \Rightarrow q) \vdash q$.

We also have

\[
AXAf = AXf, \quad AGAf = AGf.
\]
8. Modularity in MCTL*

We will exhibit formulas \( f \) such that for every \( (C; E) \) and for every \( p_1 \) we have

\[
\forall p_2; \quad [ (C; E) \models f(p_1) ] \Rightarrow [ (C; E) \models f(p_1 \parallel p_2) ],
\]

which allows to deduce

\[
\forall p_2; \quad [ p_1 \models f ] \Rightarrow [ p_1 \parallel p_2 \models f ]
\]

Such formulas are called modular.

8.1. Preliminary problem

Find the execution formulas \( f' \) such that

\[
f' \text{ exec}(p_1; p_2 \parallel p_3; C; E) \equiv (\text{or } \Rightarrow) \quad f' \text{ exec}'(p_1 \parallel p_2; p_3; C; E)
\]

for any \( p_1, p_2 \) and \( p_3 \). (These formulas are the stable ones).

If we consider nondeterministic processes, let \( \text{exec}(p_1; p_2 \parallel p_3; C; E) \) be a particular execution of \( p_1 \) in the environment \( (p_2 \parallel p_3; C; E) \) and \( \text{exec}'(p_1 \parallel p_2; p_3; C; E) \) the corresponding execution of \( p_1 \parallel p_2 \) in the environment \( (C, E) \) (making the same nondeterministic choices). We then have

\[
\begin{align*}
\text{exec}_1(p_1; p_2 \parallel p_3; C; E) &= \text{exec}'_1(p_1 \parallel p_2; p_3; C; E) \cap S(p_1), \\
\text{exec}_2(p_1; p_2 \parallel p_3; C; E) &\subseteq \text{exec}'_2(p_1 \parallel p_2; p_3; C; E), \\
\text{exec}_3(p_1; p_2 \parallel p_3; C; E) &= \text{exec}'_3(p_1 \parallel p_2; p_3; C; E), \\
\text{exec}_4(p_1; p_2 \parallel p_3; C; E) &= \text{exec}'_4(p_1 \parallel p_2; p_3; C; E).
\end{align*}
\]

We often note \( \text{exec}(p_1) \) for \( \text{exec}(p_1; p_2 \parallel p_3; C; E) \) and \( \text{exec}'(p_1 \parallel p_2) \) for \( \text{exec}'(p_1 \parallel p_2; p_3; C; E) \). We suppose \( f' \) is in normal form.

P1: (a) \( f' = s_0 \).

\[
[s_0 \text{ exec}(p_1)] \iff [s_0 \in \text{exec}_1(p_1) \, (t = 0)]
\]

\[
\Rightarrow [s_0 \in \text{exec}'_1(p_1 \parallel p_2) \, (t = 0)] \iff [s_0 \text{ exec}'(p_1 \parallel p_2)].
\]

Because \( \text{exec}_1(p_1) \, (t = 0) = \text{exec}'_1(p_1 \parallel p_2) \, (t = 0) \cap S(p_1) \). We also have the equivalence and not only the implication if \( s_0 \in S(p_1) \).

(b) \( f' = \text{ip } s_0 \).

\[
[ip \, s_0 \text{ exec}(p_1)] \iff [s_0 \in \text{exec}_3(p_1) \, (t = 0)]
\]

\[
\Leftrightarrow [s_0 \in \text{exec}'_3(p_1 \parallel p_2) \, (t = 0)] \iff [\text{ip } s_0 \, \text{exec}'(p_1 \parallel p_2)]
\]

because \( \text{exec}_3(p_1; p_2 \parallel p_3; C; E) = \text{exec}'_3(p_1 \parallel p_2; p_3; C; E) \).
(c) \( f' = \text{em } e \).

\[\text{[em } e \text{ exec}(p_1) \iff [e \in \text{exec}_2(p_1) \ (t = 0)] \]

\[\implies [e \in \text{exec}_2'(p_1 \ || p_2) \ (t = 0)] \iff [\text{em } e \text{ exec}'(p_1 \ || p_2)]\]

because \( \text{exec}_2(p_1 ; p_2 ; p_3 ; C ; E) \subseteq \text{exec}_2'(p_1 \ || p_2 ; p_3 ; C ; E) \).

The equivalence is true if \( e \) is private (also called proper) to \( p_1 \), i.e. if \( p_1 \) is the only process allowed to emit \( e \).

(d) \( f' = \text{ip } e \).

\[\text{[ip } e \text{ exec}(p_1) \iff [e \in \text{exec}_4(p_1) \ (t = 0)] \]

\[\iff [e \in \text{exec}_4'(p_1 \ || p_2) \ (t = 0)] \iff [\text{ip } e \text{ exec}'(p_1 \ || p_2)]\]

because \( \text{exec}_4(p_1 ; p_2 ; p_3 ; C ; E) = \text{exec}_4'(p_1 \ || p_2 ; p_3 ; C ; E) \).

**P2:** (a) \( f' = f \land (\forall) g \). If

\[f(g) \text{ exec}(p_1) \implies (\iff) \ f(g) \text{ exec}'(p_1 \ || p_2),\]

then

\[f \land (\forall) g \text{ exec}(p_1) \implies (\iff) f \land (\forall) g \text{ exec}(p_1 \ || p_2).\]

(b) \( f' = \neg f \). The normal form of \( f' \) being \( \neg f \), \( f \) must be an atomic proposition. Otherwise a rewriting would carry the \( \neg \) inside \( f : \neg f \rightarrow \{\neg f\} \). The only atomic propositions verifying (1) are \( f = \text{ip } s_0 \) and \( f = \text{ip } e \). Then the property (1) is true with an equivalence:

\[\neg f \text{ exec}(p_1) \iff \neg f \text{ exec}'(p_1 \ || p_2)\]

because

\[f \text{ exec}(p_1) \iff f \text{ exec}'(p_1 \ || p_2).\]

These properties are still true for \( f = s_0 \) and \( f = \text{em } e \) if \( s_0 \) is a state of \( p_1 \) and if \( e \) is private to it.

**P3:** (a) \( f' = Gf \) where \( f \) is a state formula. First consider the case \( f' = GAf \). We will note \( \text{exec}_j(p_1 ; p_2 ; C ; E ; t) \) or \( \text{exec}_j(t) \) for \( \text{exec}_j(p_1 ; p_2 ; C ; E) \).

\[\text{[GAf exec}(p_1 ; p_2 || p_3 ; C ; E)\]

\[\iff [\forall t ; (C \cup C'(t); E \cup E'(t)) \vDash Af(p_1 ; p_2 \ || p_3)]\]

\[\iff [\forall t ; (C \cup \text{exec}_3(t); E \cup \text{exec}_4(t)) \vDash Af(p_1 ; p_2 \ || p_3)]\]

\[\iff [\forall t ; \forall p_4 ; \forall \text{exec}(p_1 ; p_2 \ || p_3 || p_4 ; C \cup \text{exec}_3(t); \]

\[E \cup \text{exec}_4(t); f \text{ exec}]\]

\[\iff \ (\text{or } \Rightarrow \text{ if } f \text{ checks the property (1) with an } \Rightarrow) \]

\[\forall t ; \forall p_4 ; \forall \text{exec}''(p_1 \ || p_2 ; p_3 \ || p_4 ; C \cup \text{exec}_3'(t); \]

\[E \cup \text{exec}_4'(t); f \text{ exec}].\]
Control machines: a new model of parallelism

\[ E_t \cup \text{exec}_4(t); f \text{exec}' \]
\[ \Leftrightarrow [\forall t; (C_t \cup \text{exec}_4(t); E_t \cup \text{exec}_4(t)) = A_f(p_1 \| p_2; p_3)] \]
\[ \Leftrightarrow [G A_f \text{exec}'(p_1 \| p_2; p_3; C; E)]. \]

If \( f' = G \forall f \) the proof is all the same except at line (\( \alpha \)).

\[ [\forall t; \exists p_4; \exists \text{exec}(p_1; p_2 \| p_3 \| p_4; C_t \cup \text{exec}_3(t); E_t \cup \text{exec}_4(t)); f \text{exec}] \]
\[ \Leftrightarrow [G \forall f \text{exec}(p_1; p_2 \| p_3; C; E)]. \] \( \quad (\alpha) \)

The proofs in the case \( f' = G(f^1 \land f^2) \) and \( f' = G(f^1 \lor f^2) \) where \( f^1 \) and \( f^2 \) are state formulas are straightforward. The case \( f' = G(\neg f) \) may never occur, \( \neg f \) not being a state formula.

(b) \( f' = G f \) where \( f \) is an execution formula. We suppose that \( f \) checks the property (1). Then

\[ [G f \text{exec}(p_1; p_2 \| p_3; C; E)] \]
\[ \Leftrightarrow [\forall t; f \text{exec}(p_1; p_2 \| p_3; C_t \cup C'(t); E_t \cup E'(t)))] \]
\[ \Leftrightarrow [\forall t; f \text{exec}(p_1; p_2 \| p_3; C_t \cup \text{exec}_3(t); E_t \cup \text{exec}_4(t)))] \]
\[ \Leftrightarrow [\forall t; f \text{exec}(p_1; p_2 \| p_3; C_t \cup \text{exec}_3(t); E_t \cup \text{exec}_4(t)))] \]
\[ \Leftrightarrow (\Rightarrow) [\forall t; f \text{exec}(p_1; p_2 \| p_3; C_t \cup \text{exec}_3(t); E_t \cup \text{exec}_4(t)))] \]
\[ \Leftrightarrow [G f \text{exec}'(p_1; p_2 \| p_3; C; E)]. \]

So the formula \( G f \) checks the property (1) with an \( \Leftrightarrow \) or an \( \Rightarrow \) if \( f \) checks it with an \( \Leftrightarrow \) or an \( \Rightarrow \).

P4/5: (a/b). In these cases the proofs are similar to the previous ones.

**Theorem 8.1.** Suppose \( f \) is in normal form. The following holds:

1. \( f \text{exec}(p_1; p_2 \| p_3; C, E) \Leftrightarrow f \text{exec}(p_1; p_2 \| p_3; C, E) \)
   if \( f \) does not name \( \text{em} e \) and \( s_0 \);

2. \( f \text{exec}(p_1; p_2 \| p_3; C, E) \Rightarrow f \text{exec}'(p_1; p_2 \| p_3; C, E) \)
   if \( f \) names \( \text{em} e \) and \( s_0 \) but does name neither \( \neg \text{em} e \) nor \( \neg s_0 \). \( f \) is then said to be stable.

**Remark 8.2.** We will often keep \( f = (g \Rightarrow h) \) in this form, while its normal form is \( \{\neg g\} \lor \{h\} \). In such a case, for \( f \) to be stable, there should be no \( \neg \text{em} e \) or \( \neg s_0 \) in \( \{\neg g\} \), in other terms, there must be no \( \text{em} e \) or \( s_0 \) in \( g \) normal form. But it may name \( \neg \text{em} e \) and \( \neg s_0 \).
8.2. Modular formulas

Our initial problem was to find the modular formulas \( f \), i.e. such that for every \((C, E, p_1, p_2)\), the following property holds:

\[
[(C, E) \models f(p_1)] \Rightarrow [(C, E) \models f(p_1 \parallel p_2)].
\]

We will make an induction on the structure of \( f \).

\textbf{S2:} Suppose that \( f = g \land (\forall) h \) and that \( g \) and \( h \) are modular. Then

\[
[(C, E) \models g \land (\forall) h(p_1)] \Rightarrow [(C, E) \models g \land (\forall) h(p_1 \parallel p_2)]
\]

and \( g \land (\forall) h \) is modular. On the other hand \( \neg g(p_1) \Rightarrow \neg g(p_1 \parallel p_2) \) (or \( g(p_1 \parallel p_2) \Rightarrow g(p_1) \)) is false, even if \( g \) is modular. For example these four formulas are false:

\[
\begin{align*}
AXs_0(p_1 \parallel p_2) & \Rightarrow AXs_0(p_1), \\
AXip s_0(p_1 \parallel p_2) & \Rightarrow AXip s_0(p_1), \\
AXip e(p_1 \parallel p_2) & \Rightarrow AXip e(p_1), \\
AXem e(p_1 \parallel p_2) & \Rightarrow AXem e(p_1).
\end{align*}
\]

\textbf{S3:} \( f = Ag \) where \( g \) is stable. Then

\[
[(C, E) \models Ag(p_1)]
\]

\[
\Rightarrow [\forall p_2; \forall \text{exec}(p_1; p_2; C; E); g \text{ exec}]
\]

\[
\Rightarrow [\forall p_2; \forall p_3; \forall \text{exec}(p_1; p_2 \parallel p_3; C; E); g \text{ exec}]
\]  \hspace{1cm} (1)

\[
\Rightarrow [\forall p_2; \forall p_3; \forall \text{exec'}(p_1 \parallel p_2; p_3; C; E); g \text{ exec'}]
\]  \hspace{1cm} (2)

\[
\Rightarrow [\forall p_2; (C, E) \models Ag(p_1 \parallel p_2)].
\]

So \( [g \text{ stable}] \Rightarrow [Ag \text{ and } (C, E) \models Ag(p_1) \text{ are modular}] \). Notice that the previous proof is false if \( f = Eg \). Moreover, \( Eg \) is not modular in the general case. Consider \( p_1 \) (Fig. 8). \( p_1 \) satisfies EXs_1(p_1). Then consider \( p_2 \) (Fig. 9). Then \( \neg \text{(EXs_1)}(p_1 \parallel p_2) = AX \neg s_1(p_1 \parallel p_2) \) holds and not EXs_1(p_1 \parallel p_2). (\( s_1 \) is supposed to be private to \( p_1 \)).

**Theorem 8.3.** The formulas \( Ag; Ag \land Ah \) or \( Ag \lor Ah \) are modular if the formulas \( f \) and \( g \) are stable.
The two main points in this theorem proof are the implications (1) and (2). The first implication is true due to the fact that our formulas can only have an $A$ in the first position. The second implication is due to the fact that we are only considering stable expression formulas.

Notice that the formula $(Af \Rightarrow Ag)(p_1)$, where $Af$ and $Ag$ are modular formulas, is not modular in the general case. But either $Af(p_1)$ is true and $(Af \Rightarrow Ag)(p_1)$ can be written as $(Af \land Ag)(p_1)$, or $Af(p_1)$ is false and we can only write $\neg Af(p_1)$. Notice the difference between $(Af \Rightarrow Ag)(p_1)$ and the correct (and eventually modular) specification $A(f \Rightarrow g)(p_1)$.

9. Modularity and models of parallelism

Studying any processes satisfying the following properties

\[ s_0 \text{ exec}(p_1; p_2 || p_3; C; E) \iff s_0 \text{ exec}(p_1 || p_2; p_3; C; E) \land s_0 \in S(p_1) \]

\[ \text{ip } s_0 \text{ exec}(p_1; p_2 || p_3; C; E) \iff \text{ip } s_0 \text{ exec}(p_1 || p_2; p_3; C; E), \]

\[ \text{em } e \text{ exec}(p_1; p_2 || p_3; C; E) \Rightarrow \text{em } e \text{ exec}(p_1 || p_2; p_3; C; E), \]

and not only control machines would have led to the same modularity results. $(s_0 \in S(p_1))$ is compulsory at the first line for $A(s_0 \Rightarrow g)$ to be modular. Notice nevertheless that these four formulas are leading back to our model of environments, even if the basical automata are different: Let $S(p_j)$ be the set of $p_j$ states, i.e. the set of values that $p_j$ controls may take in the different context. $C(p_j, p_k, t)$ is the set of $p_j$ controls at $t$ in the environment $p_k$. Then

\[ C(p_j, p_k, t) \subseteq S(p_j). \quad (1) \]

The first equivalence leads to:

\[ C(p_1; p_2 || p_3; t) = C(p_1 || p_2; p_3; t) \cap S(p_1) \]

and

\[ C(p_1; p_2; t) = C(p_1 || p_2; \emptyset; t) \cap S(p_1), \]

\[ C(p_2; p_1; t) = C(p_2 || p_1; \emptyset; t) \cap S(p_2) \quad (2) \]

and

\[ C(p_1; p_2; t) \cup C(p_2; p_1; t) = C(p_1 || p_2; \emptyset; t) \cap [S(p_1) \cup S(p_2)]. \]

(1) allows us to deduce that $C(p_1 || p_2; \emptyset; t) \subseteq S(p_1 || p_2)$ and, if we suppose that $S(p_1 || p_2) \subseteq S(p_1) \cup S(p_2)$ then

\[ C(p_1 || p_2; \emptyset; t) \subseteq S(p_1) \cup S(p_2) \quad \text{and} \]

\[ C(p_1; p_2; t) \cup C(p_2; p_1; t) = C(p_1 || p_2; \emptyset; t). \quad (3) \]
(3) means that \( p_1 \) and \( p_2 \) are writing their controls in the same environments (here \( C(p_1, p_2, \emptyset, t) \)), which leads to our model of control environments. Conversely, the equation (2) means that \( p_1 \) controls are given by \( p_1 \parallel p_2 \) ones and then, by (3), by those of \( p_1 \) and \( p_2 \): (2) can be rewritten in

\[
C(p_1; p_2; t) = [C(p_1; p_2; t) \cup C(p_2; p_1; t)] \cap S(p_1)
\]

the term \( C(p_2; p_1; t) \) representing the synchronization on common states. This leads to the following theorem.

**Theorem 9.1.** Under the hypothesis that modular processes must satisfy the four previous properties (and \( S(p_1, p_2) \subseteq S(p_1) \cup S(p_2) \)), it follows that the control machine model is the only one which achieves modularity.

10. Introduction of a second quantifier

Until now, we have only allowed formulas such as

\[
[(C, E) \models Af(p_1)] \iff [\forall p_2; \forall \text{exec}(p_1, p_2, C, E); f \text{exec}],
\]

\[
[(C, E) \models Ef(p_1)] \iff [\exists p_2; \exists \text{exec}(p_1, p_2, C, E); f \text{exec}].
\]

where the quantifier ranging over the paths is the same as the one ranging over the executions. We now allow the formulas

\[
[(C, E) \models Af(p_1)] \iff [\forall p_2; \forall \text{exec}(p_1, p_2, C, E); f \text{exec}]
\]

with similar definitions for \( \text{AA} \), \( \text{EA} \), \( \text{EE} \). We may also define \( (C, E) \models \text{Af}(p_1) \) in the following way:

\[
[(C, E) \models \text{Af}(p_1)] \iff [\forall p_2; (C, E) \models \text{Ef}(p_1; p_2)]
\]

with \( [(C, E) \models \text{Ef}(p_1; p_2)] \iff [\exists \text{exec}(p_1, p_2, C, E); f \text{exec}]. \)

Notice that this \( f(p_1; p_2) \) is not the one that we used when there was a single quantifier (Section 6.2.). Then the \( f \text{exec} \) definition is the one expected; for example

\[
[X\text{Af} \text{exec}(p_1; p_2; C; E)]
\]

\[
\iff [\forall p_3; \forall \text{exec}'(p_1; p_2; p_3; C_1 \cup C'(1); E_1 \cup E'(1)); f \text{exec}'].
\]

So \( X\text{Af} \text{exec} \) is exactly the previous \( X\text{Af} \text{exec} \).

**Remarks 10.1.** (1) Consider \( p \) (Fig. 10). \( X\text{AF} \text{em} e'(p) \) holds but not \( X\text{AXA} \text{AE} \text{em} e'(p) \).

![Fig. 10.](image-url)
(2) $EA\text{em } e'$ is not modular even if $AAXE\text{em } e'$ is as shown in Fig. 11. $p'$ (Fig. 11) satisfies $EA\text{em } e'(p')$ but $EA\text{em } e'(p'||p'')$ is false in the general case. The difference between $p$ and $p'$ is that in the case of $p$ the initial $A$ (in $AAXE\text{em } e'$) enforces that $EA\text{em } e'$ is true for every $p_i$ in parallel with $p$; this is not true for $p'$.

(3) In the general case, $(C; E) \models EAf$ (or $(C; E) \models EEf$) is not modular. So we only will study the formulas beginning with an $A$.

**Theorem 10.2.** The formulas $(C; E) \models AAf$ and $(C; E) \models AEf$ are modular.

Notice that while it is forbidden to have an existential quantifier in first position, this is allowed in the following.

**Proof.** Consider the case $AEf$. We will prove $$[(C; E) \models AEf(p)] \Rightarrow [(C; E) \models AEf(p||p')]$$ as follows:

$$([(C; E) \models AEf(p)] \
\Rightarrow [\forall p_1; \exists \text{exec}(p; p_1; C; E); f \text{exec}] \
\Rightarrow [\forall p_1; \exists \text{exec}(p; p'||p_1; C; E); f \text{exec}] \
\Rightarrow [\forall p_1; \exists \text{exec}'(p||p'; p_1; C; E); f \text{exec}'] \
\Rightarrow [(C; E) \models AEf(p||p')].$$

The implication (1) is the same as the one which has been given previously for stable formulas. Suppose for example that $f = XEAg$. We denote $\text{exec}_3(1)$ for $\text{exec}_3(p; p'||p_1; C; E; 1)$. Then

$$[XEAg \text{exec}(p; p'||p_1; C; E)] \
\Rightarrow [\exists p_2; \forall \text{exec}'(p; p'||p_1||p_2; C_1 \cup \text{exec}_3(1); \ E_1 \cup \text{exec}_4(1)); g \text{exec}'']$$

which by induction hypothesis on $g$ and because $\text{exec}'$ and $\text{exec}''$ are corresponding executions implies

$$[\exists p_2; \forall \text{exec}''(p||p'; p_1||p_2; C_1 \cup \text{exec}_3(1); \ E_1 \cup \text{exec}_4(1)); g \text{exec}''] \
\Rightarrow [XEAg \text{exec}''(p||p'; p_1; C; E)],$$

where $\text{exec}$ and $\text{exec}'$ are also corresponding executions.
Remark 10.3. We define, as in Section 6.2

\[ [\text{XAAf exec}(p_1; p_2; C; E)] \]
\[ \Leftrightarrow [\forall p_3; \forall \text{exec}'(p_1; p_2 || p_3; C_1 \cup \text{exec}_3(1); E_1 \cup \text{exec}_4(1)); f \text{ exec}'] \],

where \( \text{exec}_3(1) = \text{exec}_3(p_1; p_2; C; E; 1) \). But we could also have defined

\[ [\text{XAAf exec}(p_1; p_2; C; E)] \]
\[ \Leftrightarrow [\forall p_3; \forall \text{exec}''(p_1; p_3; C_1 \cup \text{exec}_3(1); E_1 \cup \text{exec}_4(1)); f \text{ exec}'']. \]

This new definition seems to match better to our synthesis algorithm than the previous one. For example, when trying to realize A\text{AXAAf}, one may think that the obtained control machine will realize this formula with the second semantics. But in fact, for such formulas, the two semantics are the same. On the other hand, formulas such as A[GAEf \Rightarrow g] are modular with the first semantics, while not with the second.

11. Fully abstractness

In this section we will describe control machines by means of modular formulas. A somewhat similar work can be found in [7], for classical automata and classical temporal logic formulas.

In order to give a fully abstractness result (definition in [24]), we have to modify our definitions. We still define \( \text{exec}(p_1, p_2, C, E) \) as \( (\text{exec}_1, \text{exec}_2, \text{exec}_3, \text{exec}_4) \) but \( \text{exec}_1(t) \) is now the set of states reached (at \( t \)) by \( p_1 \), only by means of one of its own transitions (or by one of its initiations) and not by identification of states. \( p_1 \) writes its states in \( C_1 \). We also define a new atomic proposition, \( \text{re s}_0 \), which means that \( p_1 \) reached \( s_0 \) by itself, so \( \text{re s}_0(p_1, t) \Leftrightarrow s_0 \in C_1(t) \); \( \text{exec}_2(t) \) is still the set of all the signals emitted by \( p_1 \) at \( t \); we now write them in \( E_1 \) so \( \text{em e}(p_1, t) \Leftrightarrow e \in E_1(t) \); \( \text{exec}_3(t) \) is now the set of states reached by \( p_2 \) and \( C \) so \( \text{exec}_3(t) = C(t) \cup C_2(t) \); \( \text{exec}_4(t) \) is now the set of signals emitted by \( p_2 \) and \( E \) at \( t \) so \( \text{exec}_4(t) = E(t) \cup E_2(t) \).

So

\[ C'(t) = C(t) \cup C_1(t) \cup C_2(t) = \text{exec}_1(t) \cup \text{exec}_3(t) \]

and

\[ s_0(t) \Leftrightarrow s_0 \in C'(t) \cap S(p_1) \Leftrightarrow s_0 \in [\text{exec}_1(t) \cup \text{exec}_3(t)] \cap S(p_1). \]

Then

\[ \text{re s}_0(t) \Leftrightarrow s_0 \in \text{exec}_1(t), \]
\[ s_0(t) \Leftrightarrow s_0 \in [\text{exec}_1(t) \cup \text{exec}_3(t)] \cap S(p_1), \]
\[ \text{ip s}_0(t) \Leftrightarrow s_0 \in \text{exec}_1(t) \cup \text{exec}_3(t), \]
\[ \text{em e}_0 \Leftrightarrow e_0 \in \text{exec}_2(t), \]
\[ \text{ip e}_0(t) \Leftrightarrow e_0 \in [\text{exec}_2(t-1) \cup \text{exec}_4(t-1)]. \]
So it is possible to define the modular properties with just the new executions. We also introduce a new kind of formula, for example

\[ AG[E(s_0 \land ip e_0) \Rightarrow E(Xre s_1 \land em e_1) \land E(Xre s_2 \land em e_2)](p_1) \]

which means

\[ \forall(C; E), \forall p_2, \forall t, \exists exec(p_1; p_2; C; E), (s_0 \land ip e_0) \) exec

\[ \Rightarrow [\exists exec'(p_1; p_2; C; E), (Xre s_1 \land em e_1) \) exec'

\[ \land \exists exec''(p_1; p_2; C; E), (Xre s_2 \land em e_2) \) exec''] \].

This formula is obviously modular. It describes the transition of Fig. 12. Then we define \( \text{Prop}(p_1) \) as the set of \( p_1 \) modular properties and \([p_1]\), the observational semantics of \( p_1 \),

\[ [p_1] = \{ \exists exec|\exists(p_2; C; E), exec = exec(p_1; p_2; C; E) \} \].

![Fig. 12.](image)

**Theorem 11.1.** \( [p_1] = [p_2] \Leftrightarrow \text{Prop}(p_1) = \text{Prop}(p_2) \).

**Proof.** If \( [p_1] = [p_2] \), then the execution sets of \( p_1 \) and \( p_2 \) are equal and then \( p_1 \) and \( p \) have the same properties. So \( \text{Prop}(p_1) = \text{Prop}(p_2) \).

Conversely suppose that \( \text{Prop}(p_1) = \text{Prop}(p_2) \). We first describe \( p_1 \) and \( p_2 \) by means of modular formulas. Suppose, for example that \( p_1 \) has just two transitions (Fig. 13). We describe these two transitions by the formula

\[ AG[E(s_0 \land s_3) \Rightarrow E(Xre s_1 \land Xre s_4) \land E(Xre s_1 \land Xre s_2)

\land E(Xre s_2 \land Xre s_4) \land E(Xre s_2 \land Xre s_2)] \].

![Fig. 13.](image)
which means that the two nondeterministic choices are independent of each other. We also have

$$\text{AG}[E_{s_0} \Rightarrow E(Xre s_1) \land E(Xre s_2)],$$
$$\text{AG}[E_{s_2} \Rightarrow E(Xre s_4) \land E(Xre s_3)].$$

With three transitions we should have considered:

$$\text{AG}[E(s_0 \land s_3 \land s_6) \Rightarrow \ldots], \quad \text{AG}[E(s_0 \land s_3) \Rightarrow \ldots] \quad \text{and}$$
$$\text{AG}[E(s_0) \Rightarrow \ldots].$$

As those properties belong to $\text{Prop}(p_\sim) = \text{Prop}(p_2)$, they also belong to $\text{Prop}(p_2)$. So if there is a set of transitions from $(s_0, s_3)$ to $(s_1, s_4)$ (for example) in $p_\sim$, there is also a set of transitions in $p_2$ from $(s_0, s_3)$ to $(s_1, s_4)$ because $\text{AG}[E(s_0 \land s_3) \Rightarrow E(Xre s_1 \land Xre s_4)]$ holds. But now we must prove that in $p_2$ there is such a set of transitions which does not reach any other states or emit any signal. Otherwise, it could not be the same sets of transitions in $p_\sim$ and in $p_2$. If such a set of transitions does not exist, then $p_2$ satisfies

$$\text{AGA}[(s_0 \land s_3) \land X(s_1 \land s_4) \Rightarrow X(s_0 \lor s_3 \lor s_6 \lor \ldots \lor s_n) \lor (em e_1 \lor em e_2 \lor \ldots \lor em e_p)].$$

where $\{s_0, s_1, \ldots, s_n\}$ and $\{e_1, \ldots, e_p\}$ are the state and signal alphabets, which are supposed to be finite. But this property is false for $p_\sim$ as the transition $(s_0, s_3) \rightarrow (s_1, s_4)$ belongs to $p_\sim$. So this property does not belong to $\text{Prop}(p_2)$ and there exists a set of transitions in $p_2$ reaching no other states than $(s_1, s_4)$ (and emitting no signals.)

So we have proved that every set of transitions of $p_\sim$ also belongs to $p_2$. A similar proof holds for the initialization states. So $[p_\sim] \subseteq [p_2]$ and $[p_\sim] = [p_2]$ hold.

**Theorem 11.2.** The semantics $\text{Prop}$ is fully abstract with regard to the observational semantics $[ ]$, i.e.

$$\forall p_1, p_2, \quad \{\text{Prop}(p_1) = \text{Prop}(p_2)\}$$
$$\Leftrightarrow \{\forall \text{Cont}; [\text{Cont}(p_1)] = [\text{Cont}(p_2)]\}.$$  

where $\text{Cont}(\ )$ is a context.

**Proof.** Consider a context and call it $p_3: \text{cont}(\ ) = p_3$. We have to prove that $[p_\sim] = [p_2] \Rightarrow [p_\parallel p_3] = [p_2 \parallel p_3]$ or

$$\forall p_4, C, E; \forall \text{exec}^1(p_1 \parallel p_3; p_4; C; E), \exists (p_4; C'; E'),$$
$$\exists \text{exec}^2(p_2 \parallel p_3; p_4; C'; E'), \quad \text{exec}^1 = \text{exec}^2.$
Consider $\text{exec}^3(p_1; p_3 || p_4; C; E)$ the execution of $p_1$ corresponding to $\text{exec}^1$. As $[p_1] = [p_2]$ holds,

$$\exists (p'_1; C'; E'), \exists \text{exec}^4(p_2; p'_4; C'; E'),$$

$$\text{exec}^3(p_1; p_3 || p_4; C; E) = \text{exec}^4(p_2; p'_4; C'; E').$$

This means that in some environment $(\text{exec}^1, \text{exec}^2) = (\text{exec}^3, \text{exec}^4)$, $p_1$ and $p_2$ may have the same behavior: $(\text{exec}^1, \text{exec}^2)$. And conversely, in the same environment $(\text{exec}^3, \text{exec}^4) = (\text{exec}^1, \text{exec}^2)$, $(p_3 || p_4; C; E)$ and $(p'_4; C'; E')$ may have the same behavior, $(\text{exec}^3, \text{exec}^4) = (\text{exec}^1, \text{exec}^2)$.

So if we consider $\text{exec}^5(p_2; p_3 || p_4; C; E)$, $p_2$ may have the behavior $(\text{exec}^1, \text{exec}^2)$ provided that $(p_3 || p_4; C; E)$ has the behavior $(\text{exec}^4, \text{exec}^4)$ and conversely. Then $\text{exec}^5(p_2; p_3 || p_4; C; E)$ may be $(\text{exec}^1, \text{exec}^2, \text{exec}^3, \text{exec}^4)$. The formal proof is an induction on $t$. Then

$$\exists \text{exec}^6(p_2; p_3 || p_4; C; E),$$

$$\text{exec}^3(p_1; p_3 || p_4; C; E) = \text{exec}^6(p_2; p_3 || p_4; C; E).$$

In these two executions, $p_1$ and $p_2$ have the same behavior, $(\text{exec}^1, \text{exec}^2) = (\text{exec}^3, \text{exec}^4)$. As the environments are the same, $(C; E)$ in both cases, it is possible to enforce that $p_3$ (and $p_4$) has the same behavior in $\text{exec}^1$ and in $\text{exec}^5$. If we now consider $\text{exec}^5(p_1; p_3 || p_4; C; E)$ and $\text{exec}^5(p_2; p_3 || p_4; C; E)$, the executions corresponding to $\text{exec}^3$ and $\text{exec}^5$, then $\text{exec}^5 = \text{exec}^5$. So $[p_1 || p_3] \subseteq [p_2 || p_3]$ and $[p_1 || p_3] = [p_2 || p_3]$ hold.

### 12. Environment modularity

**Proposition 12.1.** Consider $(C_1; E_1)$ and $(C_2; E_2)$ and suppose that there exists a process $p_2$ realizing $(C_2; E_2)$ while working alone. Then

$$[(C_1; E_1) \subseteq (C_2; E_2) \text{ and } (C_1; E_1) \models f(p)] \Rightarrow [(C_2; E_2) \models f(p)].$$

We first construct $p_{2,e}$, realizing in any environment $E_2$. Consider $p_2$ and rename all its states so they are now private to it. Call $p_{2,e}$ this new control machine. As $p_2$ realizes $(C_2; E_2)$ while working alone and as $p_{2,e}$ states are private to it, $p_{2,e}$ realizes $E_2$ in any environment.

Then we construct $p_{2,e}$ always adding to its environment $C_2$. This means that in any control environment $C$, $p_2$ realizes $C_2 \cup [C \cap S(p_2)]$ (because of state identification).

For example if $p_2$ is the control machine of Fig. 14, then $p_{2,e}$ is as in Fig. 15. Notice that $p_{2,e}$ does not emit anything while $p_2$ does. The main point in the $p_{2,e}$ construction is that, if it arrives in $s_1$ (for example) by state identification, this extra control will be lost at the next moment in NIL. So $p_{2,e}$ just realizes $C_2 \cup [C \cap S(p_2)]$. 
Then consider $p_2' = p_{2,c} \parallel p_{2,e}$ and suppose, for example, that $f = \text{AE}f'$.

$$[(C_1; E_1) \models \text{AE}f'(p)] \iff [\forall p'; \exists \text{exec}(p; p'; C_1; E_1); f' \text{ exec}]$$

$$\Rightarrow [\forall p'; \exists \text{exec}'(p; p'\parallel p_2'; C_1; E_1); f' \text{ exec}']$$

$$\Rightarrow [\forall p'; \exists \text{exec}''(p; p'; C_2; E_2); f' \text{ exec}']$$

$$\Rightarrow [(C_2; E_2) \models \text{AE}f'(p)].$$

Implication (1) is true because

$$\forall \text{exec}'(p; p'\parallel p_2'; C_1; E_1); \exists \text{exec}''(p; p'; C_2; E_2); \text{ exec}' = \text{exec}''$$

because $\forall j, \text{exec}_j' = \text{exec}_j''$.

Informally, this property is still one of modularity. First consider a process $p_1$ realizing, while working alone, $(C_1; E_1)$ and a process $p$ working in the environment $p_1$. $p$ is reading and writing its controls and signals in $(C'; E')$ and $p_1$ in $(C_1; E_1)$. Recall (Section 3) that this means that $p$ has no influence on $p_1$. Then consider $p$ working in the environment $p_1$ and $p_2$. $p$ is reading and writing its controls and signals in $(C'; E')$, $p_1$ in $(C_1; E_1)$ and $p_2$ in $(C_2; E_2)$. We suppose that $(C'; E'), (C_1; E_1)$ and $(C_2; E_2)$ are physically different and that $(C'; E')$ contains copies of $(C_1; E_1)$ and $(C_2; E_2)$. So $p_1$ and $p_2$ influence $p$ but neither $p_{1(2)}$ nor $p$ influence $p_{2(1)}$. Then $p$ is working in the environment $(C_2; E_2) = (C_1; E_1) \cup (C_2; E_2)$ and the modularity property says that $f(p)$ still holds.

We defined $(\text{EnvC}; \text{EnvE})$ as environments independent of $p_1$ and $p_2$ reactions (Section 3). But notice that we could have said that only $\text{EnvE}$ was independent of $p_1$ and $p_2$ and that the environment only add $\text{EnvC}$ to $\text{EnvC}'$ (or that the environment realizes $\text{EnvC} \cup [\text{EnvC}' \cap S(p_3)]$ because of state identification). All the results would have been the same.
Proposition 12.2. With the same hypothesis on \((C; E)\) and \((C'; E')\) as on \((C_1; E_1)\) and \((C_2; E_2)\), the following holds:

\[ [(C; E) \models f(p) \land (C'; E') \models f'(p')] \Rightarrow [(C \cup C'; E \cup E') \models f \land f'(p \parallel p')] \]

Proof. Immediate. □

Definition 12.3. Let \(P\) be an environment property; \(P(C; E)\) means that the environment \((C; E)\) checks the property \(P\). We then define

\[ [P \models f(p)] \iff [\forall (C; E); P(C; E) \Rightarrow (C; E) \models f(p)]. \]

Proposition 12.4. Let \(f\) be a modular formula. Then the two following modularity properties hold:

1. \[ [P \models f(p)] \Rightarrow [\forall p'; P \models f(p \parallel p')]. \]

2. \[ [(P' \Rightarrow P) \text{ and } (P \models f(p))] \Rightarrow [P' \models f(p)]. \]

Proof. Immediate. □

13. Compilation of MCTL*

By hypothesis we suppose that a state is either always common to every process or always private to one of them. In particular a state may not be private to a process just at one moment. Releasing this restriction allows to realize more formulas but leads to a syntactic check-up over these formulas.

(1) Realization of \(As\): Cf. Fig. 16.

\(1')\) Realization of \(Aip_s\): We realize \(As\) as in (1). As we are realizing \(Af\) by induction, replacing \((Aip_s)\) by \((As)\) leads to replacing \((ip_s)\) by \((s)\) everywhere in \(f\) except in \(g\) in \((g \Rightarrow h)\) (see later).

(2) Realization of \(Aem_e\): Cf. Fig. 17.
We realize $AX^{-1}em$, in the same way than in (2). This leads to replacing $ip e$ by $X^{-1}em$ everywhere in $f$ except in $g$ in $(g \Rightarrow h)$.

Remark 13.1. Realizing (and even compositionally) $A \rightarrow s$, $Aia s$, $A \rightarrow em e$, $Aia e$ (and then $A[f \leftrightarrow g]$ too) leads to introducing private states and signals, which is out of the scope of this article. Notice that in a conventional programming language, it is also impossible to write programs realizing $A \rightarrow s_0$. Notice also that this restriction ensures that all the considered modular specifications are consistent, since our compilation algorithm is an induction one.

(3) Realization of $A(f \land g)$: By induction, we suppose that we have already constructed $CM_1$ verifying $Af$ (noted $CM_1 Af$) and $CM_2 Ag$. If $CM_1$ and $CM_2$ do not have in common a state which should be private to one of them, then $CM_1 \parallel CM_2 A(f \land g)$ holds.

(4) Realization of $AGAf$: Suppose that $CM Af$ has no private states. Then consider $AGCM$, the control machine obtained by looping on $CM$ initial states (Fig. 19) where the transition is looping on $CM$ initial states.

Example. See Fig. 20. $AGCM$ is in fact $CM \parallel AXCM \parallel \cdots \parallel AX^nCM \parallel \cdots$ and realizes

$$Af \land AXAf \land \cdots \land AX^nAf \land \cdots = AGAf$$

because $CM$ has no private states.
(5) **Realization of** $\text{AXAf}$: We have already constructed $\text{CM Af}$. Consider $\text{AXCM}$, the control machine obtained by adding a state $\alpha$ before the initial states $s_j$ of $\text{CM}$, and transitions from $\alpha$ to all the $s_j$ such that at $t = t_{cm}$ $\text{AXCM}$ is in $\alpha$ and not in the $s_j$. Then $\text{AXCM AXAf}$ holds (Fig. 21).

![Fig. 21.](image)

**Example.** $\text{CM As}$:

![CM As](image)

and $\text{AXCM AXAs}$:

![AXCM AXAs](image)

(6) **Realization of** $\text{AFAf}$: We already have constructed $\text{CM Af}$. Consider $\text{AFCM}$, the control machine obtained by adding a state $\alpha$ before the initial states $s_j$ of $\text{CM}$ and transitions from $\alpha$ to all the $s_j$, and looping on $\alpha$ before executing the transition from $\alpha$ to all the $s_j$. $\text{AFCM}$ is initialized at $t = t_{CM} - 1$ and realizes $\text{AFAf}$ (Fig. 22).

**Example.**

$\text{CM Af}$:

![CM Af](image)

and $\text{AFCM}$:

![AFCM](image)

(7) **Realization of** $A(f \lor g)$: We first define $f \lor g$. Let $f$ and $g$ be execution formulas. Then $f \lor g \Leftrightarrow N(Af \lor Ag)$. Notice that $A(f \lor g) \Rightarrow A(f \lor g)$. The difference between $\lor$ and $\wedge$ is that, with $\lor$, at $t = 0$, we already know if, in the considered execution, $f$ will be realized, or $g$. This is not granted with $\lor$.

We already have constructed $\text{CM}_1 \text{Af}$ and $\text{CM}_2 \text{Ag}$. Then if, for example, $t_{CM_1} = t_{CM_2} = 0$, $\text{CM}_1 \lor \text{CM}_2$ obtained by means of a nondeterministic choice at $t = -1$ realizes $A(f \lor g)$ (Cf. Fig. 23).

![Fig. 22.](image)
(8) **Realization of** $A(f \Rightarrow g)$: Notice that we use different algorithms for $A(\neg f \lor g)$ and $A(f \Rightarrow g)$. We will only consider the case where $f$ names only $ip$ $s_0$, $ia$ $s_0$, $ip$ $e$, $ia$ $e$ and where $f$ ends before $g$ begins. ($f$, the cause, is entirely before $g$, the effect). For $f$ ending before $g$ begins, $f$ may only use the following operators: $X$, $\lor$, $\land$, $\neg$. (Notice that specifications such as $f \lor (10s)$, i.e. $f$ until ten seconds, are just a special case of the previous case).

(a) We first define $T(f)$, the greatest instant concerned by $f$.

$$T(ip \ s) = T(ip \ e) = 0, \quad T(Xf) = 1 + T(f),$$

$$T(f \lor g) = T(f \land g) = \text{Sup}(T(f); \ T(g)), \quad T(\neg f) = T(f),$$

(b) Then we realize $CM \ A[f \leftrightarrow X^{T(f)+1}s]$ where $s$ is a compounded state (noted c.s.) with $c.s. = s_0[\text{c.s.} \lor \text{c.s.}] \land \text{c.s.}$ where $s_0$ is a classical state. The states constitutive of $s$ are private to $CM$.

- $f = ip \ s_0$ (ia $s_0$), (cf. Fig. 24).
- $f = Xf'$. We already have constructed $CM \ A[f' \leftrightarrow X^{T(f')\ +\ 1}s]$. Consider $AXCM$: the following properties hold:

$$AXCM \ AXA[f' \leftrightarrow X^{T(f')\ +\ 1}s], \quad AXCM \ AXC[f' \leftrightarrow X^{T(f')\ +\ 2}s],$$

$$AXC[f' \leftrightarrow X^{T(Xf')\ +\ 1}s], \quad AXC[f' \leftrightarrow X^{T(f')\ +\ 1}s],$$

$$AXC[f \leftrightarrow X^{T(f')\ +\ 1}s].$$

![Diagram](image-url)
\( f = f_1 \land (\forall) f_2 \). We have already constructed

\[
\text{CM}_1 A[f_1 \Leftrightarrow X^{T(f_1)+1} s_1], \quad \text{CM}_2 A[f_2 \Leftrightarrow X^{T(f_2)+1} s_2].
\]

Suppose that \( T(f) = \text{Sup}(T(f_1), T(f_2)) = T(f_1) \). We now construct \( \text{CM}'_2 A[f_2 \Leftrightarrow X^{T(f)+1} s_2^{f_2}] \). \( \text{CM}'_2 = \text{CM}_2 \parallel \text{CM}'_2 \) where \( \text{CM}'_2 \) is the machine depicted in Fig. 25. Then

\[
\text{CM}_1 \parallel \text{CM}'_2 A[f_1 \land f_2 \Leftrightarrow X^{T(f)+1}(s_1 \land s_2^{T(f)_1})] \quad \text{and} \quad \text{CM}_1 \parallel \text{CM}'_2 A[f_1 \lor f_2 \Leftrightarrow X^{T(f)+1}(s_1 \lor s_2^{T(f)_1})]
\]

hold.

(c) We will now realize \( \text{CM}' A[X^{T(f)+1} s \Rightarrow g] \). We have supposed that \( g \) begins after \( f \) ends. More precisely, suppose that \( g = X^{T(f)+2} g' \) (notice that \( g = X^{T(f)+1} e \) could be authorized too) and consider \( \text{CM}_1 A g' \). Then \( \text{CM}_2 \) is the control machine of Fig. 26 where the transition arrives on \( \text{CM}_1 A g' \) initial states and realizes \( A[ip s \Rightarrow Xg'] \). \( \text{CM}' \) is the machine of Fig. 27 where the last transition arrives on \( \text{CM}_1 A g' \) initial states and realizes

\[
A[X^{T(f)+1} ip s \Rightarrow X^{T(f)+2} g'] = A[X^{T(f)+1} ip s \Rightarrow g].
\]

(d) We have now constructed

\[
\text{CM} A[f \Leftrightarrow X^{T(f)+1} s] \quad \text{and} \quad \text{CM}' A[X^{T(f)+1} ip s \Rightarrow g].
\]

Then \( \text{CM} \parallel \text{CM}' \) realizes \( A[f \Rightarrow g] \).

(e) The previous construction allows to realize \( \text{AG}[f \Rightarrow g] (A(xt)^w(f \Rightarrow g)) \) easily. One checks easily that, although \( s \) is private to \( \text{CM} A[f \Leftrightarrow X^{T(f)+1} s] \), the control machine \( \text{AG} \) realizes \( \text{AG}[f \Leftrightarrow X^{T(f)+1} s] \) \( (A(xt)^w \text{CM} \text{ realizes } A(xt)^w[f \Leftrightarrow X^{T(f)+1} s]) \). This property is due to the fact that \( \text{CM} \) is in \( s \) only at \( t = T(f) + 1 \) and

\[
\text{CM}_1 \quad \text{Ag'}
\]
then that:

$$\forall n, (CM \parallel AX''CM) (A[f \leftrightarrow X^{T(f)+1}s] \land AX''[f \leftrightarrow X^{T(f)+1}s])$$

holds. Then as $AG\,CM'$ realizes $AG[X^{T(f)+1}ip \Rightarrow s \Rightarrow g]$, $AG(CM \parallel CM')$ realizes $AG[f \Rightarrow g]$.

(f) Remarks. (1) In [22], we realized $A[f \Rightarrow g]$ by means of $CM, A[f \Rightarrow G \neg \text{stop}] \parallel CM, A[G \text{\textbar\textbar} \text{stop} \Rightarrow g]$ with stop private to $CM$. This allows us to realize $A[f \Rightarrow g]$ in nearly all the cases, even when $f$ does not end before $g$ begins. But, because stop is private to $CM$, this method is not suitable for realizing $AG[f \Rightarrow G \neg \text{stop}]$ and $AG[f \Rightarrow g]$. In the same work, we also gave an algorithm realizing $A[f \Rightarrow g]$.

(2) If the formula $g$ begins before $f$ ends but does not use the operator $\lor$ and $\land$ (deterministic control machine), then it is possible to find $f'$ and $g'$ such that $CM, A[f \Rightarrow g] \leftrightarrow CM \land, A(f' \Rightarrow g')$ and that $g'$ begins after $f'$ ends. $f'$ is the beginning of $f$ until $t$. $g'$ is the part of $g$ controlled by $f'$. Then we can realize $CM, A[f' \Rightarrow g']$ and $CM = \parallel, CM, \text{realizes } A[f \Rightarrow g]$. But this algorithm has a quadratic complexity.

(9) Realization of $A(f \lor g)$: (a) If $CM, A[f]$ and $CM, A[g]$ may be constructed, and if for example $t_{CM1} = t_{CM2} = 0$, the control machine of Fig. 28 realizes $A(f \lor g)$.

(b) If $g$ names only $ip \ s_0, \ ia \ s_0, \ X, \land$ and $\neg$, then we have already constructed $CM, A(f \Rightarrow g) \leftrightarrow CM \land, A(f \Rightarrow g')$ and that $g'$ begins after $f$ ends. $f$ is the beginning of $f$ until $t$. $g'$ is the part of $g$ controlled by $f$. Then we can realize $CM, A[f' \Rightarrow g']$ and $CM = \parallel, CM, \text{realizes } A[f \Rightarrow g]$. But this algorithm has a quadratic complexity.

$$AG[\neg g \leftrightarrow X^{T(g)+1}s], \quad AX^{T(g)+1}\alpha, \quad AG[\alpha \land ip \Rightarrow Xf' \land X\alpha],$$

and one deduces easily that $AGCM \parallel CM, A[f \lor g]$ holds. Notice that, once more, although $s$ is private to $CM$, $\forall n, (CM \parallel AX''CM) (A[f \lor g] \land AX''[f \lor g])$ holds, which allows to realize $AG[f \lor g]$ (in the same way we could realize $A(xt)^{\lor}(f \lor g)$).

![Fig. 28](image-url)

![Fig. 29](image-url)
(10) **Realization of $AXf$, $AGf$ and $AFf$:** We have already realized $AXAf$, $AGAf$ and $AFAf$. And $AXAf \leftrightarrow AXf$ and $AGAf \leftrightarrow AGf$ hold. In the $AFf$ case, only the implication holds: $AFAf \Rightarrow AFf$. For example, the control machine of Fig. 30 realizes $AFX^2s_2$ but not $AFAF^2s_2$. Nevertheless, the previous implication allows to realize $AFf$ by means of CM $AFAf$. Notice too that if $f = Xf'$ then $AFf = AFXf' = AXFf' = AAXAf f'$ and that if $f = Ff'$ then $AFf = AFFf' = AFf'$.

(11) **Generalization to regular operators:** Following Wolper [45], consider the operator $(xt)^w (t: \text{true}; x: \text{undefined})$.

$$ [A(xt)^w f(p_1)] \Leftrightarrow [\forall (C, E); \forall p_2; \forall \text{exec}(p_1, p_2, C, E); \forall t; f \text{exec}_{t+1}]. $$

The control machine $CM_0$ (Fig. 31) realizes $A(xt)^w T$ and is better than $CM'_0$ which realizes $A(xt)^w T$ but also $AG(T \Rightarrow X^T)$. This second property can be troublesome if there is another process $CM_2$ in parallel with $CM'_0$ (Fig. 32). $CM_2$ realizes $AT \wedge AXT$. $CM'_0 \parallel CM_2$ realizes $A(xt)^w T \wedge (AT \wedge AXT)$. But $CM'_0 \parallel CM_2$ realizes $AGT$ and not only $A(xt)^w T \wedge (AT \wedge AXT)$.

Suppose now that there exists $CM_1$ realizing $Af$ without any private state. For example $CM_1 AXs$ (Fig. 33) where by $\alpha$ we mean the initial state of $CM_1$. Then $AG(\alpha \Rightarrow Xs) \Rightarrow \Rightarrow$ holds. More generally, let the $\alpha_j$ be the initial states of $CM_1$ (we suppose that $k_{CM1} = 0$). $CM_1 AG[\wedge \_ \_ \alpha_j \Rightarrow f]$ holds because $CM_1$ has no private state. Now suppress $CM_1$ initial states (suppress the property $CM_1 Af$ but not $CM_1 AG[\wedge \_ \_ \alpha_j \Rightarrow f]$).
and identify $T$ and $\bigwedge_j \alpha_j: \bigwedge_j \alpha_j = T$. Then $CM_0 \parallel CM_1$ satisfies $A(xt)^w f$ (Cf. Fig. 34) because

$$CM_0 A(xt)^w T, \quad CM_1 AG(\bigwedge_j \alpha_j \Rightarrow f), \quad \bigwedge_i \alpha_i = T,$$

$$\Rightarrow CM_0 \parallel CM_1 A(xt)^w f.$$

Notice that we could also have introduced and synthesized easily an explicit "loop" operator.

Finally, let us mention the strict specifications. Roughly speaking, a specification $f$ is a strict specification of process $p$ if every execution of $p$ (in every environment) does nothing else than what $f$ describes or as little as possible [22]. For example, in Section 13(11), $CM_0$ does not realize strictly $A(xt)^w T$, because $CM_0 \parallel CM_2$ realizes $AG T$ and not only $A(xt)^w T \land (AT \land AXT)$. This, of course, may be very troublesome for applications. Nevertheless, notice that this problem exists even in the classical parallelism model, although it is more acute in the control machine one, as controls may be created by other processes. It seems to be possible to compile automatically processes realizing strictly given specifications [22] while classical model checkers never prove that the checked process realizes strictly its specification.

(12) **Complexity:** The compile time, number of states, number of transitions, environment size and execution time needed to synthesize a control machine from a given specification $f$, by means of the previous algorithm, are obviously linear in the size of $f$.

14. Hierarchical networks and specifications

In Section 3 we have already shown how $(EnvC; EnvE)$ could be used for hierarchies of processes. Nevertheless, notice that we achieved modularity only within the same level. For example, in Section 3 $f(p_1) \land g(p_2) \Rightarrow f \land g(p_1 \parallel p_2)$ holds but not $f(p_1) \land h(p_3) \Rightarrow f \land h(p_1 \parallel p_3)$, because $p_3$ is not reading its controls in the
same environment as $p_1$. Suppose for example that $f = A[s_0 \Rightarrow Xs_1]$ and $h = A[Xs_1 \Rightarrow X^2s_2]$. If $p_1$ is in $s_0$ at $t = 0$, then it will be in $s_1$ at $t = 1$ but not $p_3$ so $X^2s_2$ is not granted.

We now show how hierarchies of processes may be introduced in the same level of our model, which achieves modularity even within these hierarchies.

Consider the example of Fig. 35. In $CM_0$, the states $s_1$ and $s_2$ must be understood rather as processes than as states. The previous diagram is then a representation of a hierarchical network. Consider also the CM specification: $AG[Xs_1 \land X^2s_2]$. In this specification, $s_1$ and $s_2$ must be understood rather as specifications of $CM_1$ and $CM_2$ than as states. The $CM_1$ and $CM_2$ specifications are $AG(s_j \Rightarrow X_{t_j})$, therefore we may replace $s_j$ by $s_j \land X_{t_j}$. CM specification is then $AG[X(s_1 \land X_{t_1}) \land X^2(s_2 \land X_{t_2})]$. Notice that $CM_0$ just creates controls to initialize other processes (control machine).

**Remark.** If we want $CM_1$ and $CM_2$ not to be able to influence $CM_0$, we just have to duplicate the $CM_0$ states as in Section 12.
15. Conclusions

We have given a new model of parallelism allowing a new kind of communication and synchronization between processes. We defined a specification language, a temporal logic, with a new semantics. This new semantics and the model lead to modular (compositional) specifications naming internal states, which allow proofs of parallel programs, easy debugging, natural program development and a clear and easy programming methodology. Then we compiled the main part of these modular specifications and this compilation is linear both in size and time. So this specification language is in fact a logical programming language efficiently compilable. As in every logical programming language, the problem of program verification has disappeared. Once a specification has been written, the compiler constructs automatically a control machine realizing this specification. Moreover, we proved that for any process \( p \), the set of modular specifications of \( p \) is a fully abstract semantics of \( p \), which means that the modular specifications are enough for describing a process entirely. Finally we showed two different ways in which our model may be used for hierarchical networks and specifications.

We believe that the modular programming methodology underlying this work is closer to the designer intuition and easier to use than many others. We also feel that our compositional semantics (and our model of computation) offers a nice and practical alternative to the classical partial order semantics \([46, 13, 5, 8]\) interleaving semantics \([29, 35]\) or compositional semantics \([25, 41, 23]\).

We would like to emphasize that our proof system, being purely modular, can be used for processes working in parallel with an arbitrary number of others processes \([40]\).

In addition we would like to draw attention to the fact that the interval between two instants does not have to be constant and may be, for example, externally driven. Thus we could have introduced temporal operators such as \( X_e \), meaning "at the next instant when signal \( e \) will be present". This allows the definition of numerous different clocks as in \([6]\) and leads to real-time applications. We can also model the control flow of concurrent processes, either directly or introducing asynchrony \([30]\). Then the operator \( X \) could mean, for example: "the next time when all the processes have stopped working and are ready to communicate (synchronize)".

Related works in the field of real-time systems include Esterel \([6]\) and Statecharts \([21]\). Both provide a mechanism allowing chain reaction effects but lead to temporal paradoxes similar to the one pictured in Section 3 and give rise to serious semantical difficulties. The main problem is that these models, allow in their very heart, nonmodular specifications. Nevertheless, such a chain reaction mechanism is very powerful and highly desirable for real-time applications. For example, a watch counting milliseconds must be able to go from state \( 099 \) ms to state \( 100 \) ms at once, i.e. executing the two transitions from \( 9 \) to \( 0 \) and the one from \( 0 \) to \( 1 \) at the same moment, even if the third one is induced by the second one, itself induced by the first one. Such a mechanism, somewhat restricted, may also be provided in the
control machine model with minor changes. And, what is more, it does not lead to temporal paradoxes or nonmodular specifications.

Acknowledgment

We would like to thank André Arnold, Jeoffroy Beauquier, Jean-Marc Steyaert and specially Maurice Nivat for their interest in this work and their comments on earlier drafts of this paper.

References


