Modular Compilation of a Synchronous Language

Annie Ressouche, Daniel Gaffé, Valérie Roy

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Modular Compilation of a Synchronous Language

Daniel Gaffé — Annie Ressouche — Valérie Roy

N° 6424

January 24, 2008

Thème COG

ISSN 0249-6399 ISRN INRIA/RR--6424--FR+ENG

Modular Compilation of a Synchronous Language

Daniel Gaffé [*] , Annie Ressouche [†] , Valérie Roy [‡] [§]

Thème COG — Systèmes cognitifs
Projet Pulsar

Rapport de recherche n° 6424 — January 24, 2008 — 61 pages

Abstract: Synchronous languages rely on formal methods to facilitate the developement of applications in an efficient and reusable way. In fact, formal methods have been advocated as a means of increasing the reliability of systems, especially those which are safety or business critical. It is even more difficult to develop automatic specification and verification tools due to limitations such as state explosion, undecidability, etc... In this work, we design a new specification model based on a reactive synchronous approach. We benefit from a formal framework well suited to perform compilation and formal validation of systems. In practice, we design and implement a special purpose language (LE) with two semantic: its *behavioral semantic* helps us to define a program by the set of its behaviors and avoid ambiguity in programs interpretation; its *equational semantic* allows the compilation of programs into software and hardware targets (C code, Vhdl code, Fpga synthesis, Model checker input format). Our approach is relevant with respect to the two main requirements of critical realistic applications: modular compilation allows us to deal with large systems, while model-based approach provides us with formal validation. There is still a lack of efficient and modular compilation means for synchronous languages. Despite of relevant attempts to optimize generated code, no approach considers modular compilation. This report tackles this problem by introducing a compilation technique which relies on the equational semantic to ensure modularity completed by a new algorithm to check causality cycles in the whole program without checking again the causalty of sub programs.

Key-words: synchronous language, modular compilation, behavioral semantic, equational constructive semantic, modularity, separate compilation.

[*] I3S Laboratory and CNRS
[†] INRIA Sophia Antipolis
[‡] CMA ENM Sophia Antipolis
[§] thanks to S. Moisan and J.P Rigault for their careful reading and their fruitful suggestions

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Compilation modulaire d'un langage synchrone

Résumé : Dans ce rapport, nous étudions le développement de systèmes critiques. Les méthodes formelles se sont avérées un moyen efficace pour augmenter la fiabilité de tels systèmes, en particulier ceux qui requièrent une certaine sécurité de fonctionnement. Neanmoins, le développement d'outils automatiques de spécification et de vérification est limité entre autre par la taille des modèles formels des systèmes ou par des problèmes d'indécidabilité. Dans ce travail, nous définissons un langage réactif synchrone (LE) dédié à la spécification de systèmes critiques. Ce faisant, nous bénéficions d'un cadre formel sur lequel nous nous appuyons pour compiler séparement et valider les applications. Plus précisement, nous définissons deux sémantiques pour notre langage: une *sémantique comportementale* qui associe à un programme l'ensemble de ses comportements et évite ainsi toute ambiguité dans l'interpretation des programmes. Nous définissons aussi une *sémantique équationnelle* dirigeant la compilation de programmes vers différentes cibles (code c, code vhdl, synthétiseurs fpga, observateurs), permettant ainsi de traiter des applications logicielles et matérielles et aussi de les valider. Notre approche est pertinente vis à vis des deux principales exigences de réelles applications critiques: la compilation modulaire permet de traiter des applications conséquentes et l'approche formelle permet la validation. On peut constater que le domaine des langages synchrones manque encore de méthodes pour compiler les programmes de façon efficace et modulaire. Bien sur, certaines approches optimisent les codes produits d'un facteur important, mais aucune d'entre elles n'envisagent une compilation modulaire.

Mots-clés : langage synchrone, compilation modulaire , sémantique comportementale sémantique constructive equationnelle, modularité, compilation séparée.

Contents

1 Introduction

We address the design of safety-critical control-dominated systems. By design we mean all the work that must be done from the initial specification of a system to the embedding of the validated software into its target site. The way control-dominated systems work is *reactive* in the sense of D. Harel and A. Pnueli definition[11]: they react to external stimuli at a speed defined and controlled by the system's environment. The evolution of a reactive system is a sequence of reactions raised by the environment. A control-dominated application can then be naturally decomposed into a set of communicating reactive sub-systems each dealing with some specific part of the global behavior, combined together to achieve the global goal.
It is now stated that general purpose programming languages are not suited to design reactive systems: they are clearly inefficient to deal with the inherent complexity of such systems. From now on, the right manner to proceed is to design languages dedicated to reactive systems. To this aim, *synchronous languages* such as Esterel[3] and SyncCharts [1], dedicated to specify event-driven applications; Lustre and Signal[9], data flow languages well suited to describe signal processing applications like, have been designed. They are model-based languages to allow formal verification of the system behavior and they agree on three main features:

1. *Concurrency*: they support functional concurrency and they rely on notations that express concurrency in a user-friendly manner. LE adopts an imperative Esterel-like style to express parallelism. However, the semantic of concurrency is the same for all synchronous languages and simultaneity of events is primitive.

2. *Simplicity*: the language formal models are simple (usually mealy machines or netlists) and thus formal reasoning is made tractable. In particular, the semantic for parallel composition is clean.

3. *Synchrony*: they support a very simple execution model. First, memory is initialized and then, for each input event set, outputs are computed and then memory is updated. Moreover, all mentioned actions are assumed to take finite memory and time.

Synchronous languages rely on the *synchronous hypothesis* which assumes a discrete logic time scale, made of instants corresponding to reactions of the system. All the events concerned by a reaction are simultaneous: input events as well as triggered output events. As a consequence, a reaction is instantaneous (we consider that a reaction takes no time), there are no concurrent partial reactions, and determinism is thus ensured.
There are numerous advantages to the synchronous approach. The main one is that temporal semantic is simplified, thanks to the afore mentioned logical time. This leads to clear temporal constructs and easier time reasoning. Another key advantage is the reduction of state-space explosion, thanks again to discrete logical time: systems evolve in a sequence of discrete steps, and nothing occurs between two successive steps. A first consequence is that program debugging, testing, and validating is easier. In particular, formal verification of synchronous programs is possible with techniques like model checking. Another consequence is that synchronous language compilers are able to generate automatically embeddable code, with performances that can be measured precisely.

Although synchronous languages have begun to face the state explosion problem, there is still a need for further research on their efficient and modular compilation. The initial compilers translated the program into an extended finite state machine. The drawback of this approach is the potential state explosion problem. Polynomial compilation was first achieved by a translation to equation systems that symbolically encode the automata. This approach is successfully used for hardware synthesis and is the core of commercial tools [15] but the generated software may be very slow. Then several approaches translate the program into event graphs [16] or concurrent data flow graphs [7, 13] to generate efficient C code. All these methods have been used to optimize the compilation times as well as the size and the execution of the generated code.
However none of these approaches consider a modular compilation. Some attempts allow a distributed compilation of programs [16, 7] but no compilation mechanism relies on a modular semantic of programs. Of course there is a fundamental contradiction in relying on a formal semantic to compile reactive systems because a perfect semantic would combine three important properties: *responsiveness*, *modularity* and *causality*. Responsiveness means that we can deal with a logical time and we can consider that output events occur in the same reaction as the input events causing them. It is one of the foundations of the synchronous hypothesis. Causality means that for each event generated in a reaction, there is a causal chain of events leading to this generation; no causality loop may occur. A semantic is modular when "environment to component" and "component to component" communication are treated symmetrically. In particular, the semantic of the composition of two reactive systems can be deduced from the respective semantic of each sub-part. Another aspect of modularity is the coherent view each subsystem has of what is going on. When an event is present, it is broadcasted all around the system and is immediately available for every part which listens to it. Unfortunately, there exists a theorem ("the RMC barrier theorem") [12] that states that these three properties cannot hold together in a semantic. Synchronous semantic are responsive and modular. But causality remains a problem in these semantic and modular compilation must be completed by a global causality checking.
In this paper we introduce a reactive synchronous language, we define its behavioral semantic that gives a meaning to programs and an equational semantic allowing first, a modular compilation and, second, a separate verification of properties. Similarly to other synchronous semantic, we must check that programs have no potential causality loop. As already mentioned, causality can only be checked globally since a bad causality may be created when performing the parallel composition of two causal sub programs. We compile LE programs into equation systems and the program is causal if its compilation is cycle free. The major contribution of our approach relies on the introduction of a new sorting algorithm that allows us to start from already compiled and checked subprograms to compile and check the overall program without sorting again all the equations.

2 LE Language

LE language belongs to the family of reactive synchronous languages. It is a discrete control dominated language. We first describe its syntax (the overall grammar is detailed in appendix A).
The LE language unit are *named modules*. The language's operators and constructions are chosen to fit the description of reactive applications as a set of concurrent communicating sub-systems.

Communication takes place between modules or between a module and its environment. Sub-system communicates via *events*.
The *module interface* declares the set of *input events* it reacts to and the set of *output events* it emits. For instance, the following piece of code shows the declarative part of a *Control* module used in the example in section 6.

```
module Control:
Input:forward, backward, upward,  downward, StartCycle;
Output:MoveFor, MoveBack, MoveDown, SuckUp, EndCycle ;
```

2.1 LE Statements

The *module body* is expressed using a set of *control operators*. They are the cornerstone of the language because they operate over event's status. Some operators terminate instantaneously, some other takes at least one instant. We mainly distinguish two kinds of operators: usual programming language operators and operators devoted to deal with logical time.

2.1.1 Non Temporal Statements

LE language offers two basic instructions:

- The *nothing instruction* does "nothing" and terminates instantaneously.
- The *event emission instruction* (*emit speed*) sets to present the status of the emitted signal.

Moreover, some operators help us to built composite instructions:

- The *present-then-else instruction* (*present S { P1} else { P2}*) is a usual conditional statement except that boolean combinations of signals status are used as conditions.
- In the *sequence instruction* ($P_1 \gg P_2$) the first sub-instruction P_1 is executed. Then, if P_1 terminates instantaneously, the sequence executes immediately its second instruction P_2 and stops whenever P_2 stops. If P_1 stops, the sequence stops. The sequence terminates at the same instant as its second sub-instruction P_2 terminates. If the two sub-instructions are instantaneous, the sequence terminates instantaneously.
- The *parallel instruction*($P_1 \| P_2$) begins the execution of its two sub-instructions at the same instant. It terminates when both sub-instructions terminate. When the two sub-instructions are instantaneous, the parallel is instantaneous. Notice that the parallel instruction agrees with the synchronous hypothesis and allows the simultaneity of trigger signals causing P_1 or P_2.
- A *strong or weak preemption instruction* over a signal S can surround an instruction P as in: $abort\ P\ when\ S$. While the signal status evaluates to "absent", instruction P continues its execution. The instant the event evaluates to "present", the instruction is forced to terminate. When the instruction is preempted, the `weak` preemption let the instruction ends its current execution while the `strong` one does not. If the instruction terminates normally without been preempted, the preemption instruction also terminates and the program execution continues.

- A *Loop instruction* ($loop\ \{P\}$) surrounds an instruction P. Instruction P is automatically restarted the same instant it terminates. The body of a loop cannot be instantaneous since it will start again the execution of its body within the same instant.
- *Local signals instruction* ($local\ S\ \{P\}$) is used to encapsulate communication channels between two sub systems. The scope of S is restricted to P. As a consequence, each local signal tested within the body of the local instruction must be emitted from the body.
- A *module call instruction*(Run) is used to run an external module inside another module. Recursive calls of module are not allowed. Running a module does not terminate instantaneously. In the declarative part of the module, you can specify the paths where the already compiled code of the called modules are:

```
Run: "./TEST/control/" : Temporisation;
Run: "./TEST/control/" : NormalCycle;
```

2.1.2 Temporal Statements

There are two temporal operators in LE .

- The *pause instruction* stops for exactly one reaction.
- The *waiting instruction* (*wait S*) waits the presence of a signal. The first time the execution of the program reaches a `wait` instruction, the execution stops (whatever the signal status is). At the beginning of the following instant, if the signal status is tested "present" the instruction terminates and the program continues its execution, otherwise it stays stopped.

2.1.3 Automata Specification

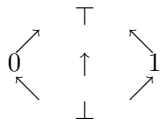
Because it remains difficult to design an automaton-like behavior using the previously mentioned operators, our language offers an *automaton description* as a native construction. An automata is a set of states and labeled transitions between states. Some transitions are initial and start the automata run while terminal states indicate that the automaton computation is over. The label of transitions have two fields: a trigger which is a boolean combination of signal status and an output which is the list of signals emitted when the transition is taken (i.e when the trigger part is true). LE automata are Mealy machines and they have a set of input signals to define transition triggers and a set of output signals that can be emitted when a transition is raised. In LE , the body of a module is either an instruction or an automaton. It is not allowed to build new instructions by combining instructions and automata. For instance, the only way to put in parallel an automaton and the emission of a signal is to call the module the body of which is the automata through a run operation. Practically, we offer a syntactic means to describe an automaton (see appendix A for a detailed syntax). Moreover, our graphical tool (GALAXY) helps users edit automata and generate the LE code.

3 LE Behavioral Semantic

LE behavioral semantic is useful to give a meaning to each program and thus to define its behavior without ambiguity. To define the behavioral semantic of LE, we first introduce a logical context to represent events, then we define the LE process calculus in order to describe the behavioral semantic rules.

3.1 Mathematical Context

Similarly to others synchronous reactive languages, LE handles *broadcasted signals* as communicating means. A program reacts to input events by producing output events. An *event* is a signal carrying some information related to its *status*. The set of signal status ξ ($\xi = \{\bot, 0, 1, \top\}$) [1] is intented to record the status of a signal at a given instant. Let S be a signal, S^x denotes its instant current status. More precisely, S^1 means that S is present, S^0 means that S is absent, $S^\bot$ means that S is neither present nor absent and finally $S^\top$ corresponds to an event whose status cannot be induced because it has two incompatible status in two different sub parts of the program. For instance, if S is both absent and present, then it turns out to have $\top$ status and thus an error occurs. Indeed. the set ξ is a complete lattice with the $\leq$ order:



Composition Laws for ξ

We define 3 internal composition laws in ξ: $\boxplus$, $\boxdot$ and $\neg$ (to extend the usual operations defined for classical boolean set $\mathbb{B}$), as follows:
The $\boxplus$ law is a binary operation whose result is the upper bound of its operands:

$\boxplus$	1	0	$\top$	$\bot$
1	1	$\top$	$\top$	1
0	$\top$	0	$\top$	0
$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	1	0	$\top$	$\bot$

Particularly:

- $\bot \boxplus \bot = \bot$;
- $1 \boxplus 0 = 0 \boxplus 1 = \top$;

[1] we also denote true and false values of ξ boolean algebra by 1 and 0 by misuse of language. Nevertheless, when some ambiguity could occur, we will denote them 1_ξ, 0_ξ.

- $\top$ is an absorbing element;

The $\boxdot$ law is a binary operation whose result is the lower bound of its operands:

$\boxdot$	1	0	$\top$	$\bot$
1	1	$\bot$	1	$\bot$
0	$\bot$	0	0	$\bot$
$\top$	1	0	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Particularly:

- $\top \boxdot \top = \top$;
- $1 \boxdot 0 = 0 \boxdot 1 = \bot$;
- $\bot$ is an absorbing element;

Finally, the $\neg$ law is an inverse law in ξ:

x	$\neg x$
1	0
0	1
$\top$	$\bot$
$\bot$	$\top$

The set ξ with these 3 operations verifies the axioms of *Boolean Algebra*: commutative and associative axioms for $\boxplus$ and $\boxdot$, distributive axioms both for $\boxdot$ over $\boxplus$ and for $\boxplus$ over $\boxdot$, neutral elements for $\boxplus$ and $\boxdot$ and complementarity.

Commutativity:	$x \boxplus y = y \boxplus x$	$x \boxdot y = y \boxdot x$	(1)
Associativity:	$(x \boxplus y) \boxplus x = x \boxplus (y \boxplus z)$	$(x \boxdot y) \boxdot x = x \boxdot t(y \boxdot z)$	(2)
Distributivity:	$x \boxdot (y \boxplus z) = (x \boxdot y) \boxplus (x \boxdot z)$	$x \boxplus (y \boxdot z) = (x \boxplus y) \boxdot (x \boxplus z)$	(3)
Neutral elements:	$x \boxplus \bot = x$	$x \boxdot \top = x$	(4)
Complementarity:	$x \boxplus \neg x = \top$	$x \boxdot \neg x = \bot$	(5)

Axioms (1) and (4) are obvious looking at the previous tables that define the $\boxplus$ and $\boxdot$ laws. Axioms (2) and (4) are also obviously true but their proofs necessitate to compute the appropriate tables. Finally, axiom (5) results from the following table:

x	$x \boxplus \neg x$	$x \boxdot \neg x$
1	$1 \boxplus 0 = \top$	$1 \boxdot 0 = \bot$
0	$0 \boxplus 1 = \top$	$0 \boxdot 1 = \bot$
$\top$	$\top \boxplus \bot = \top$	$\top \boxdot \bot = \bot$
$\bot$	$\bot \boxplus \top = \top$	$\bot \boxdot \top = \bot$

As a consequence, ξ **is a Boolean algebra** and the following theorems are valid:

Identity law:	$x \boxplus x = x$	$x \boxdot x = x$
Redundancy law:	$x \boxplus (x \boxdot y) = x$	$x \boxdot (x \boxplus y) = x$
Morgan law:	$\neg (x \boxplus y) = \neg x \boxdot \neg y$	$\neg (x \boxdot y) = \neg x \boxplus \neg y$
Neutral element:	$x \boxplus \top = \top$	$x \boxdot \bot = \bot$

In such a setting, xor, nor, $nand$, $\Leftrightarrow$, $\Rightarrow$ are defined:

$$
\begin{array}{lcl}
x \; xor \; y & = & x \boxdot \neg y \boxplus y \boxdot \neg x \\
x \; nor \; y & = & \neg x \boxdot \neg y \\
x \; nand \; y & = & \neg x \boxplus \neg y \\
x \Leftrightarrow y & = & (\neg x \boxdot \neg y) \boxplus (x \boxdot y) \\
x \Rightarrow y & = & \neg x \boxplus y
\end{array}
$$

Hence, we can apply these classical results concerning Boolean algebras to solve equation systems whose variables belong to ξ. For instance, the equational semantic detailed in section 4 relies on boolean algebra properties to compute signal status as solution of status equations.
Moreover, since ξ is a lattice, the $\boxplus$ and $\boxdot$ operations are monotonic: let x, y and z be elements of ξ, $(x \leq y) \Rightarrow (x \boxplus z \leq y \boxplus z)$ and $(x \leq y) \Rightarrow (x \boxdot z \leq y \boxdot z)$.

Condition Law

We introduce a *condition* law (**◄**) in ξ to drive a signal status with a boolean condition:

$$
\begin{array}{ccc}
\xi \times \mathbb{B} & \longrightarrow & \xi \\
(x, c) & \longmapsto & x \blacktriangleleft c
\end{array}
$$

This law is defined by the following table:

x	c	$x \blacktriangleleft c$
1	0	$\bot$
0	0	$\bot$
$\top$	0	$\bot$
$\bot$	0	$\bot$
1	1	1
0	1	0
$\top$	1	$\top$
$\bot$	1	$\bot$

This condition law allows us to change the status of an event according to a boolean condition. It will be useful to define both LE behavioral and equational semantic since the status of signals depend of the termination of the instructions that compose a module. Intuitively, a signal keeps its status if the condition is true, otherwise its status is set to $\bot$.

Relation between ξ and $\mathbb{B}^2$

ξ is bijective to $\mathbb{B} \times \mathbb{B}$. We define the following encoding:

signal status	encoding
1	11
0	10
$\top$	01
$\bot$	00

Hence, a signal status is encoded by 2 boolean variables. The first boolean variable of the status of a signal (S) is called its definition (S_{def}), while the second one is called its value (S_{val}). According to the encoding law, when $S_{def} = 0$ the signal S has either $\top$ or $\bot$ value for status and it is not defined as present or absent. On the opposite, when $S_{def} = 1$, the signal is either present or absent. It is why we choose to denote the first boolean projection of a signal status by S_{def}.
$\mathbb{B}$ is the classical boolean set with 3 operators *and* (denoted .), *or* (denoted +) and *not* (denoted $\overline{x}$, for boolean x). According to the previous encoding of ξ into $\mathbb{B} \times \mathbb{B}$ and after algebraic simplification, we have the following equalities related to $\boxplus$, $\boxdot$ and $\neg$ operators. Let X and Y be 2 elements of ξ:

$$
\begin{array}{lcl}
(X \boxplus Y)_{def} & = & X_{def}.\overline{Y_{def}}.\overline{Y_{val}} + Y_{def}.\overline{X_{def}}.\overline{X_{val}} + (X_{def}.Y_{def}).\overline{(X_{val} \oplus Y_{val})} \\
(X \boxplus Y)_{val} & = & X_{val} + Y_{val}
\end{array}
$$

$$
\begin{array}{lcl}
(X \boxdot Y)_{def} & = & X_{def}.\overline{Y_{def}}.Y_{val} + Y_{def}.\overline{X_{def}}.X_{val} + (X_{def}.Y_{def}).\overline{(X_{val} \oplus Y_{val})} \\
(X \boxdot Y)_{val} & = & X_{val}.Y_{val}
\end{array}
$$

$$
\begin{array}{lcl}
(\neg X)_{def} & = & X_{def} \\
(\neg X)_{val} & = & \overline{X_{val}}
\end{array}
$$

$$
\begin{array}{lcl}
(X \blacktriangleleft c)_{def} & = & X_{def}.c \\
(X \blacktriangleleft c)_{val} & = & X_{val}.c
\end{array}
$$

where $\oplus$ is the *exclusive or* operator of classical boolean set. The proof of the last equality is detailed in appendix D.
On the opposite side, we can expand each boolean element into a status member, 0 correspond to 0, and 1 to 1. More precisely let x be an element of $\mathbb{B}$ and $\xi(x)$ its corresponding status, then $\xi(x)_{def} = 1$ and $\xi(x)_{val} = x$.

Notion of Environment

An *environment* is a finite set of events. Environments are useful to record the current status of signals in a reaction. Thus a signal has a unique status in an environment: if S^x and s^y belongs to the same environment, then $x = y$.
We extend the operation defined in ξ to environments. Let E and E' be 2 environments:

$$\begin{array}{lcl} E \boxplus E' & = & \{S^z | \exists S^x \in E, S^y \in E', z = x \boxplus y\} \\ E \boxdot E' & = & \{S^z | \exists S^x \in E, S^y \in E', z = x \boxdot y\} \\ \neg E & = & \{S^{\neg x} | \exists S^x \in E\} \\ E \blacktriangleleft c & = & \{S \blacktriangleleft c \mid S \in E\} \end{array}$$

We define a relation ($\preceq$) on environments as follows:

$$E \preceq E' \text{ iff } \forall S^x \in E, \exists S^y \in E' | S^x \leq S^y$$

Thus $E \preceq E'$ means that E is included in E' and that each element of E is less than an element of E' according to the lattice order of ξ. As a consequence, the $\preceq$ relation is a total order on environments and $\boxplus$ and $\boxdot$ operations are monotonic according to $\preceq$.
Finally, we will denote $E^{\top}$, the environment where all events have $\top$ status.

3.2 LE Behavioral Semantic

In order to describe the behavioral semantic of LE , we first introduce a process algebra associated with the language. Then we can define the semantic with a set of rewriting rules that determines a program execution. The semantic formalize a reaction of a program P according to an event input set. $P \xmapsto[E]{E'} P'$ has the usual meaning: E and E' are respectively input and output environments; program P reacts to E, reaches a new state represented by P' and the output environment is E'. To compute such a reaction we rely on the behavioral semantic of LE . This semantic supports a rule-based specification to describe the behavior of each operator of LE process algebra associated with LE language. A rule has the form: $p \xrightarrow[E]{E', TERM} p'$ where p and p' are elements of LE process algebra. E is an environment that specifies the status of the signals declared in the scope of p, E' is the output environment and $TERM$ is a boolean flag true when p terminates. This notion of termination differs from the one used in Esterel language successive behavioral semantic. It means from the current reaction, p is able to terminate and this information will be sustained until the real termination occurs.
Let P be a LE program and p its corresponding process algebra term. Given an input event set E, a reaction is computed as follows:

$$P \xmapsto[E]{E'} P' \quad \text{iff} \quad p \xrightarrow[E]{E',\, TERM} p'$$

LE Process Calculus (PLE)

The PLE process algebra associated to LE language is defined as follows:

- `nothing`;
- `halt`;
- `!s` (emit `s`);

- `wait s`;
- `iwait s` (wait immediate `s`);
- `s` ? p : q (present `s` $\{p\}$ else $\{q\}$);
- $p \| q$;
- $p \gg q$;
- $p \uparrow$`s` (abort $\{p\}$ when `s`);
- $p*$ (loop {p});
- $p \backslash$`s` (local `s` $\{p\}$);
- $\mathcal{A}(\mathcal{M}, \mathcal{T}, \mathcal{C}ond, M_f, \mathcal{O}, \lambda)$. Automata $\mathcal{A}$ is a structure made of 6 components:
  1. a finite set of *macro states* ($\mathcal{M}$). Each macro state M may be is itself composed of a sub term p (denoted $M[p]$);
  2. a finite set of *conditions* ($\mathcal{C}ond$);
  3. a finite set of *transitions* ($\mathcal{T}$). A transition is a 3-uple $< M, c, M' >$ where $c \in \mathcal{C}ond$ is a boolean condition raising the transition from macro state M to macro state M'. We will denote $M \rightarrow M'$ for short in the rest of the report and $c_{M \rightarrow M'}$ will denote the condition associatesd with the transition. . $\mathcal{T}$ is also composed of initial transitions of the form: $\rightarrow M'$. They are useful to start the automata run. When condition c is true, the macro state M' is reached;
  4. a *final* macro state M_f;
  5. a finite set of *output signals* ($\mathcal{O}$) paired with an *output function* λ that links macro states and output signals: $\lambda : \mathcal{T} \longrightarrow \mathcal{P}(\mathcal{O})$, defined as follows: $\lambda(M \rightarrow M') = o \subseteq \mathcal{O}$ is the set of output signals emitted when the trigger condition $c_{M \rightarrow M'}$ is true.

Each instruction of LE has a natural translation as an operator of the process algebra. As a consequence, we associate a term of the process algebra with the body of each program while the interface part allows to build the global environment useful to define the program reaction as a rewriting of the behavioral semantic. Notice that the operator `iwait s` does not correspond to any instruction of the language, it is introduced to express the semantic of the `wait` statement. It is a means to express that the behavior of a term takes at least one instant. It is the case of `wait s` that skip an instant before reacting to the presence of `s`.
More precisely, we introduce a mapping: Γ : LE $\rightarrow$ PLE , which associates a PLE term with each LE program. Γ is defined according to the syntax of the LE language.
Let P be a LE program, $\Gamma(P)$ is structurally defined on the body of P.

- $\Gamma(\text{nothing}) =$ `nothing`;
- $\Gamma(\text{halt}) =$ `halt`;

- $\Gamma(\text{emit } \mathtt{s}) = !\mathtt{s}$;
- $\Gamma(\text{wait } \mathtt{s}) = \mathtt{wait\ s}$;
- $\Gamma(\text{present}\mathtt{s}P_1 \text{ else } P_2) = \mathtt{s}?\Gamma(P_1) : \Gamma(P_2)$;
- $\Gamma(P_1 \| P_2) = \Gamma(P_1) \| \Gamma(P_2)$;
- $\Gamma(P_1 \gg P_2) = \Gamma(P_1) \gg \Gamma(P_2)$;
- $\Gamma(\text{abort } P_1 \text{ when } \mathtt{s}) = \Gamma(P_1) \uparrow \mathtt{s}$;
- $\Gamma(\text{loop } \{P_1\}) = \Gamma(P_1)*$;
- $\Gamma(\text{local } \mathtt{s}\{P_1\}) = \Gamma(P_1) \backslash \mathtt{s}$;
- $\Gamma(\text{run } P_1) = \text{wait } \mathtt{tick} \gg \Gamma(P_1)$ where `tick` is a "clock" signal present in each reaction;
- $\Gamma(\mathcal{A}(\mathcal{M}, \mathcal{T}, \mathcal{C}ond, M_f, \mathcal{O}, \lambda) = \mathcal{A}(\mathcal{M}, \mathcal{T}, \mathcal{C}ond, M_f, \mathcal{O}, \lambda)$.

Behavioral Semantic Rules

The basic operators of LE process algebra have the following rewriting rules. Both `nothing` and `halt` have no influence on the current environment, but the former is always ready to leave and the latter never. The emit operator is ready to leave and the signal emitted is set present in the environment [2].

$$\mathtt{nothing} \xrightarrow[E]{E,\,1} \mathtt{nothing} \qquad (nothing)$$

$$\mathtt{halt} \xrightarrow[E]{E,\,0} \mathtt{nothing} \qquad (halt)$$

$$!\mathtt{s} \xrightarrow[E]{E[\mathtt{s}\leftarrow 1],\,1} \mathtt{nothing} \qquad (emit)$$

Wait

The semantic of `wait` is to wait at least one instant. Thus, to express its behavior, we introduce the `iwait` operator. Then, `wait s` is not ready to leave, and rewrites into `iwait s`. This rewriting behaves like `wait s` except that it reacts instantaneously to the signal presence.

$$\mathtt{wait\ s} \xrightarrow[E]{E,\,0} \mathtt{iwait\ s} \qquad (wait)$$

$$\frac{\mathtt{s}^1 \in E}{\mathtt{iwait\ s} \xrightarrow[E]{E,\,1} \mathtt{nothing}} \quad (iwait1) \qquad \frac{\mathtt{s}^1 \notin E}{\mathtt{iwait\ s} \xrightarrow[E]{E,\,0} \mathtt{iwait\ s}} \quad (iwait2)$$

[2] In the following, we will denote $\mathtt{s} \leftarrow 1$ the setting of s'value to 1 ($\xi(\mathtt{s}) = 1$)

Present

The semantic of $\mathtt{s}\ ?\ p\ :\ q$ operator depends on the status of $\mathtt{s}$ in the initial environment E. If $\mathtt{s}$ is present (resp absent) in E, the operator behaves like p (resp q) (rules $present1$ and $present2$). Otherwise, if $\mathtt{s}$ is undefined we cannot progress in the rewriting system (rule $present3$) and if the computation of $\mathtt{s}$ internal status results in $\top$, it is an error and this last is propagated (each event is set to error in the environment).

$$\frac{p \xrightarrow[E]{E_p,\ TERM_p} p',\ q \xrightarrow[E]{E_q,\ TERM_q} q',\ \mathtt{s}^1 \in E}{\mathtt{s}\ ?\ p\ :\ q \xrightarrow[E]{E_p,\ TERM_p} p'} \quad (present1)$$

$$\frac{p \xrightarrow[E]{E_p,\ TERM_p} p',\ q \xrightarrow[E]{E_q,\ TERM_q} q',\ \mathtt{s}^0 \in E}{\mathtt{s}\ ?\ p\ :\ q \xrightarrow[E]{E_q,\ TERM_q} q'} \quad (present2)$$

$$\frac{p \xrightarrow[E]{E_p,\ TERM_p} p',\ q \xrightarrow[E]{E_q,\ TERM_q} q',\ \mathtt{s}^\perp \in E}{\mathtt{s}\ ?\ p\ :\ q \xrightarrow[E]{E,\ 0} \mathtt{s}\ ?\ p\ :\ q} \quad (present3)$$

$$\frac{p \xrightarrow[E]{E_p,\ TERM_p} p',\ q \xrightarrow[E]{E_q,\ TERM_q} q',\ \mathtt{s}^\top \in E}{\mathtt{s}\ ?\ p\ :\ q \xrightarrow[E]{E^\top,\ 1} \mathtt{s}\ ?\ p\ :\ q} \quad (present4)$$

Parallel

The parallel operator computes its two arguments according to the broadcast of signals between both sides and it terminates when both sides do.

$$\frac{p \xrightarrow[E]{E_p,\ TERM_p} p' \quad , \quad q \xrightarrow[E]{E_q,\ TERM_q} q'}{p \| q \xrightarrow[E]{E_p \boxplus E_q,\ TERM_p.TERM_q} p' \| q'} \quad (parallel)$$

Sequence

The sequence operator has the usual behavior. While the first argument does not terninate we don't begin the computation of the second argument (rule $sequence1$). When it terminates , we start the second argument (rule $sequence2$).

$$\frac{p \xrightarrow[E]{E_p,\ 0} p'}{p \gg q \xrightarrow[E]{E_p,\ 0} p' \gg q} \quad (sequence1) \qquad \frac{p \xrightarrow[E]{E_p,\ 1} \mathtt{nothing}, q \xrightarrow[E_p]{E_q,\ TERM_q} q'}{p \gg q \xrightarrow[E]{E_q,\ TERM_q} q'} \quad (sequence2)$$

Abort

The behavior of the abort operator first derives the body of the statement. Thus, if the aborting signal is present is the input environment, then the statement rewrites in `nothing` and terminates (rule $abort1$). If it is not, the body of the statement is derived again (rules $abort2$ and $abort3$)

$$\frac{p \xrightarrow[E]{E_p, TERM_p} p', \mathtt{s}^1 \in E}{p \uparrow \mathtt{s} \xrightarrow[E]{E_p,\ 1} \mathtt{nothing}} \quad (abort1)$$

$$\frac{p \xrightarrow[E]{E_p,1} \mathtt{nothing}, \mathtt{s}^1 \notin E}{p\uparrow_{\mathtt{s}} \xrightarrow[E]{E_p,1} \mathtt{nothing}} \quad (abort2) \qquad \frac{p \xrightarrow[E]{E_p,0} p', \mathtt{s}^1 \notin E}{p\uparrow_{\mathtt{s}} \xrightarrow[E]{E_p,0} p'\uparrow_{\mathtt{s}}} \quad (abort3)$$

Loop

Loop operator never terminates and $p*$ behaves as $p \gg p*$.

$$\frac{p \xrightarrow[E]{E_p,0} p'}{p* \xrightarrow[E]{E_p,\ 0} p' \gg p*} \quad (loop)$$

Local

Local operator behaves as an encapsulation. Local signals are no longer visible in the surrounding environment.

$$\frac{p \xrightarrow[E\cup \mathtt{S}^{\perp}]{E_p,\ TERM_p} p'}{p\backslash \mathtt{s} \xrightarrow[E]{E_p - \{\mathtt{S}\}, TERM_p} p'\backslash \mathtt{s}} \quad (local)$$

Automata

Automata are deterministic (i.e $\forall M \in \mathcal{M},\ \exists! M \rightarrow M' \in \mathcal{T}$ such that $c_{M\rightarrow M'} = 1$).
The semantic of automata terms relies on macro state semantic. A macro state does not terminate within a single reaction. Its duration is at least one instant. Thus, $M[p]$ waits an instant and then has the same behavior than p.

$$M[p] \xrightarrow[E]{E,\ 0} p$$

If the macro state is only a state without sub term p, then

$$M \xrightarrow[E]{E,\ 0} nothing$$

Now, we define the rewriting rules for automata $\mathcal{A}$. The evaluation of a condition $c \in \mathcal{C}ond$ depends on the current status of signals in the environment. To denote the current value of a condition we will use the following notation: $E \models c = b$
Axiom:

$$\frac{\rightarrow M \in \mathcal{T}, E \models c_{\rightarrow M} = 1}{\mathcal{A} \xrightarrow[E]{E[s \leftarrow 1 | s \in \lambda(\rightarrow M)],\ 0} < \mathcal{A}, M, M[p] >} \quad (automata0)$$

Rewriting rules for automata describe the behavior of a reaction as usual. Thus, we define rewriting rules on a 3-uple: $< \mathcal{A}, M, p >$. The first element of the tuple is the automaton we consider, the second is the macro state we are in, and the third is the current evaluation of the sub term involved in this macro state.

$$\frac{p \xrightarrow[E]{E_p, TERM} p', \quad \forall M' \text{ such that } M \rightarrow M' \in \mathcal{T}\ E_p \models \sum c_{M \rightarrow M'} \neq 1}{< \mathcal{A}, M, p > \xrightarrow[E]{E_p,\ 0} < \mathcal{A}, M, p' >} \quad (automata1)$$

$$\frac{\exists M' \text{ such that } M \rightarrow M' \in \mathcal{T} \text{ and } E \models c_{M \rightarrow M'} = 1}{< \mathcal{A}, M, p > \xrightarrow[E]{E[s \leftarrow 1 | s \in \lambda(M \rightarrow M')],\ 0} < \mathcal{A}, M', M'[p] >} \quad (automata2)$$

$$\frac{p \xrightarrow[E]{E_p, 1} nothing}{< \mathcal{A}, M_f, p > \xrightarrow[E]{E_p,\ 1} nothing} \quad (automata3)$$

Rule $automata0$ is the axion to start the evaluation of the automaton. Rule $automata1$ expresses the behavior of $\mathcal{A}$ automata when all the transition trigger conditions are false: in such a case, the sub term associated with the current macro state is derived (whatever the derivation is) and the automata does not terminate. On the opposite side, rule $automata2$ expresses the automata behavior when a transition condition becomes true. In such a case, the automata steps to the next macro state specified in the condition and the emitted signals associated with the transition are set to 1 in the environment. Finally, rule $automata3$ is applied when the evaluation of the term included in the final macro state is over; then the automata computation is terminated.
The behavioral semantic is a "macro" semantic that gives the meaning of a reaction for each term of the LE process algebra. Nevertheless, a reaction is the least fixed point of a micro step semantic that computes the output environment from the initial one. According to the fact that the $\boxplus$ and $\boxdot$ operations are monotonic with respect to the $\preceq$ order, we can rely on the work about denotational semantic [8] to ensure that for each term, this least fixed point exists. Practically, we have $p \xrightarrow[E]{E'} p'$ if there is a sequence of micro steps semantic:

$$p \xrightarrow[E]{E_1} \!\!\!\!\rightarrow p_1, p_1 \xrightarrow[E_1]{E_2} \!\!\!\!\rightarrow p_2, ...$$

At each step $E_{i+1} = F^i(E_i)$, since the F_i functions are some combinations of $\boxplus$ operator and $\blacktriangleleft$ condition law, they are monotonic and then $\forall i, E_{i+1} \preceq F^i(E_i)$. Then, we have $E' = \sqcup_n F^n(E_n)$, thus it turns out that E' is the least fixpoint of the family of F^n functions. But, ξ boolean algebra is a complete lattice, then so is the set of environments, as a consequence such a least fixpoint exits.

4 LE Equational Semantic

In this section, we introduce a constructive circuit semantic for LE which gives us a practical means to compile LE programs in a modular way.
The behavioral semantic describes how the program reacts in an instant. It is logically correct in the sense that it computes a single output environment for each input event environment when there is no causality cycles. To face this causality cycle problem specific to synchronous approach, constructive semantic have been introduced [2]. Such a semantic for synchronous languages are the application of constructive boolean logic theory to synchronous language semantic definition. The idea of constructive semantic is to "forbid self-justification and any kind of speculative reasoning replacing them by a fact-to-fact propagation". In a reaction, signal status are established following propagation laws:

- each input signal status is determined by the environment;
- each unknown signal S becomes present if an "emit S" can be executed;
- each unknown signal S becomes absent if an "emit S" cannot be executed;
- the then branch of a test is executed if the signal test is present;
- the then branch of a test is not executed if the signal cannot be present;
- the else branch of a test is executed if the signal test is absent;
- the else branch of a test is not executed if the signal test cannot be absent;

A program is *constructive* if and only if fact propagation is sufficient to establish the presence or absence of all signals.
An elegant means to define a constructive semantic for a language is to translate each program into a constructive circuit. Such a translation ensures that programs containing no cyclic instantaneous signal dependencies are translated into cycle free circuits. Usually, a boolean sequential circuit is defined by a set of wires W, a set of registers R, and a set of boolean equations to assign values to wires and registers. W is partitioned into a set of input wires I, output wires O and a set of local wires. The circuit computes output wire values from input wires and register values. Registers are boolean memories that feed back the circuit. The computation of circuit outputs is done according to a propagation law and to ensure that this propagation leads to logically correct solutions, a constructive value propagation law is supported by the computation.

Constructive Propagation Law

Let $\mathcal{C}$ be a circuit, I its input wire set, R_v a register valuation (also called a "state") and w a wire expression. Following [2], the constructive propagation law has the form : $I, R_v \vdash w \hookrightarrow b$, b is a boolean value and the law means that under I and R assumptions, w evaluates to b. The definition of the the law is:

$$
\begin{array}{lll}
I, R_v \vdash b \hookrightarrow b & & \\
I, R_v \vdash w \hookrightarrow b & \text{if} & I(w) = b \\
I, R_v \vdash w \hookrightarrow b & \text{if} & R(w) = b \\
I, R_v \vdash w \hookrightarrow b & \text{if} & w = e \in \mathcal{C}, I, R_v \vdash e \hookrightarrow b \\
I, R_v \vdash w \hookrightarrow b & \text{if} & w = \overline{e}, I, R_v \vdash e \hookrightarrow \overline{b} \\
I, R_v \vdash w \hookrightarrow 1 & \text{if} & w = e + e', I, R_v \vdash e \hookrightarrow 1 \text{ or } I, R_v \vdash e' \hookrightarrow 1 \\
I, R_v \vdash w \hookrightarrow 0 & \text{if} & w = e + e', I, R_v \vdash e \hookrightarrow 0 \text{ and } I, R_v \vdash e' \hookrightarrow 0 \\
I, R_v \vdash w \hookrightarrow 1 & \text{if} & w = e.e', I, R_v \vdash e \hookrightarrow 1 \text{ and } I, R_v \vdash e' \hookrightarrow 1 \\
I, R_v \vdash w \hookrightarrow 0 & \text{if} & w = e.e', I, R_v \vdash e \hookrightarrow 0 \text{ or } I, R_v \vdash e' \hookrightarrow 0
\end{array}
$$

The $\hookrightarrow$ propagation law is the logical characterization of constructive circuits. Nevertheless, this notion also supports two equivalent characterizations. The denotational one relies on three-values boolean ($\mathbb{B}_\perp = \{\perp, 0, 1\}$) and a circuit $\mathcal{C}$ with n wires, input wire set I and registers R is considered as a monotonic function $\mathcal{C}(I, R) : \mathbb{B}_\perp^n \longrightarrow \mathbb{B}_\perp^n$. Such a function has a least fixed point and this latter is equal to the solution of the equation system associated to the logical point of view. On the other hand, the electric characterization uses the inertial delay model of Brozowski and requires electric stabilization for all delays. In [14], it is shown that a circuit $\mathcal{C}$ is constructive for I and R if and only if for any delay assignment, all wires stabilize after a time t. The resulting electrical wire values are equal to logical propagation application results.

4.1 Equational Semantic Foundations

LE circuit semantic associates a specific circuit with each operator of the language. This circuit is similar to sequential boolean circuits except that wire values are elements of ξ boolean algebra. As a consequence, the equation system associated with such a circuit handles ξ valued variables. As already mentioned, solutions of equation system allow to determine all signal status .
To express the semantic of each statement in LE , we generate a circuit whose interface handles the following wires to propagate information and so to ensure synchronization between statements.

- *SET* to propagate the control (input wire);
- *RESET* to propagate reinit (input wire);
- *RTL* ready to leave wire to indicate that the statement can terminate in the reaction (or in a further one);

Wires used to synchronize sub programs are never equal to $\perp$ or $\top$. They can be considered as boolean and the only values they can bear are true or false. Thus, according to our translation from

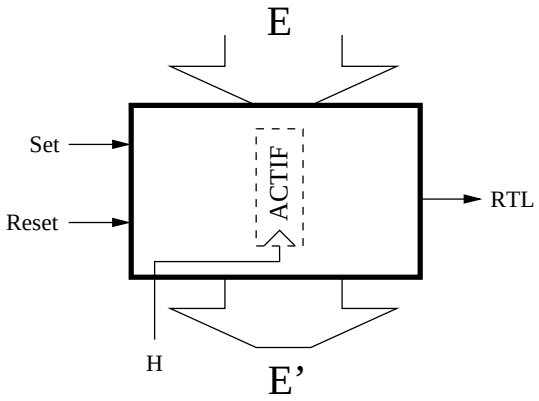


Figure 1: Circuit semantic for a LE statement

ξ to $\mathbb{B} \times \mathbb{B}$: $\mathrm{SET}_{def} = \mathrm{RESET}_{def} = \mathrm{RTL}_{def} = 1$. In the following, we will denote P_S the set of synchronization wires of P. Moreover, for statements that do not terminate instantaneously, a register is introduced (called *ACTIF*). Similarly to control wires, $\mathrm{ACTIF}_{def} = 1$. We will denote P_R the set of registers of a program P.
In order to define the equational semantic, we introduce an operator: $\boxtimes$ that acts on the element of ξ whose boolean definition value is 1. Let $\xi_{\mathbb{B}} = \{x \in \xi | x_{def} = 1\}$:

$$\begin{array}{rccc} \boxtimes : & \xi \times \xi_{\mathbb{B}} & \longrightarrow & \xi_{\mathbb{B}} \\ & (x, y) & \longrightarrow & (1, x_{def}.x_{val}.y_{val}) \end{array}$$

This new operation will be useful to define the product between a real ξ valued signal and a synchronization wire or register. It is different from $\blacktriangleleft$ operation, since this latter defines a "mux" operation and not a product.
In addition, we introduce a $\mathcal{P}re$ operation on environment in order to express the semantic of operators that do not react instantaneously. It allows to memorize all the status of current instances of events. As already said, an environment is a set of events, but circuit semantic handles wider environments than behavioral semantic. In the latter, they contain only input and output events, while in equational semantic they also contain event duplication and wires and registers. Let E be an environment, we denote $E\restriction_I$ the input events of E and $E\restriction_O$ the output ones.

$$\mathcal{P}re(E) = \{S^{\perp} | S^x \in E,\ S \notin E\restriction_I,\ S \notin E\restriction_O\} \cup \{S^x_{pre} | S^x \in E,\ S \notin E\restriction_I,\ S \notin E\restriction_O\}$$

The $\mathcal{P}re(E)$ operation consists in a duplication of events in the environment. Each event S^x is recorded in a new event S^x_{pre} and the current value of signal S is set to $\perp$ in order to be refined in the current computation. But, $\mathcal{P}re(E)$ operation does not concern interface signals because it is useless, only their value in the current instant is relevant. Moreover, this operation updates the registers values: we will denote ACTIF^+ the value of the register ACTIF computed for the next reaction.

In LE equational semantic, we consider ξ-circuits i.e circuits characterized by a set of ξ-wires, a set of ξ-registers and an environment E where ξ-wires and ξ-registers have associated ξ values. The ξ-circuit schema is described in figure 1. The ξ-circuit [3] associated with a statement has an input environment E and generates and output environment E'. The environment include input, output, local and register status.
We rely on the general theory of boolean constructiveness previously detailed. Let $\mathcal{C}$ be a ξ-circuit, we translate $\mathcal{C}$ into a boolean circuit. More precisely, $\mathcal{C} = (W, R, E)$ where W is s set of ξ-wires and R a set of ξ-registers. E is composed of a set of equations of the form $x = e$ in order to compute a status for wires and registers.
Now, we translate $\mathcal{C}$ into the following boolean circuit $\mathcal{C}^B = (W^B, R^B, D^B)$ where W^B is a set of boolean wires, R^B a set of boolean registers and D^B a set of boolean equations.

$$\begin{aligned} W^B &= \{w_{def}, w_{val} \mid w \in W\} \\ R^B &= \{w_{def}, w_{val} \mid w \in R\} \\ D^B &= \{w_{def} = e_{def}, w_{val} = e_{val} \mid w = e \in E\} \end{aligned}$$

e_{def} and e_{val} are computed according to the algebraic rules detailed section 3.1.
Now we define the constructive propagation law ($\rightsquigarrow$) for ξ-circuits. Let $\mathcal{C}$ be a ξ-circuits with $I \subseteq E$ as input wire set and $R \subseteq E$ as register set, the definition of the constructive propagation law for $\mathcal{C}$ is:

$$E \vdash w \rightsquigarrow bb \Leftrightarrow I^B, R^B \vdash w_{def} \hookrightarrow bb_{def} \text{ and } I^B, R^B \vdash w_{val} \hookrightarrow bb_{val}.$$

This definition is the core of the equational semantic. We rely on it to compile LE programs into boolean equations. Thus, we benefit from BDD representation and optimizations to get an efficient compilation means. Moreover, we also rely on BDD representation to implement a separate compilation mechanism.
Given P a LE statement. Let $\mathcal{C}(P)$ be its associated circuit [4] and E be an input environment. A reaction for the circuit semantic corresponds to the computation of an output environment composition of E and the synchronization equations of P. We denote $\bullet$ this composition operation:

$$E' = E \bullet \mathcal{C}(P) \text{ if and only if } E \cup \mathcal{C}(P) \vdash w \rightsquigarrow bb, \text{ and } E'(w) = bb, \forall w \in E \cup \mathcal{C}(P).$$

Now, we define the circuit semantic for each statement of LE . We will denote: $\langle P \rangle_E$ the output environment of P built from E input environment.

4.2 Equational Semantic of LE Statements

Nothing

The circuit for `nothing` is described in figure 9(a) in appendix E. The corresponding equation system is the following:

$$\langle \text{nothing} \rangle_{\text{E}} = \text{E} \bullet \{\text{RTL} = \text{SET}\}$$

[3] in what follow, when no ambiguity remains, we will omit the ξ prefix when speaking about ξ-circuit.
[4] the equations defining its SET, RESET and RTL wires and the equations defining its registers when it has some

Halt

The circuit for `halt` is described in figure 9(b) in appendix E. The statement is never ready to leave instantaneously.

$$\langle \text{halt} \rangle_{\text{E}} = \text{E} \bullet \{\text{RTL} = 0\}$$

Emit

The `emit` S statement circuit is described in figure 10 in appendix E. As soon as the statement receive the control, it is ready to leave. RTL and SET wires are equal and the emitted signal S is present in the output environment. We don't straightly put the value of S to 1 in the environment, we perform a $\boxplus$ operation with 1 in order to keep the possible value $\top$ and then transmit errors. Moreover, the latter is driven with the boolean value of RTL wire:

$$\langle \text{emit } S \rangle_{\text{E}} = (\text{E}[S \leftarrow (1 \boxplus \xi(S))]) \blacktriangleleft \text{RTL}_{val} \bullet \{\text{RTL} = \text{SET}\}$$

Pause

The circuit for `pause` is described in figure 11(a) in appendix E. This statement does not terminate instantaneously, as a consequence a register is created and a $\mathcal{P}re$ operation is applied to the output environment:

$$\langle \text{pause} \rangle_{\text{E}} = \mathcal{P}re(\text{E}) \bullet \left\{ \begin{array}{lcl} \text{RTL} & = & \text{ACTIF} \\ \text{ACTIF}^{+} & = & (\text{SET} \boxplus \text{ACTIF}) \boxtimes \neg\text{RESET} \end{array} \right\}$$

Wait

The circuit for `wait` is described in figure 11(b) in appendix E. The `wait` S statement is very similar to the `pause` one, except that the ready to leave wire is drive by the presence of the awaited signal:

$$\langle \text{wait } S \rangle_{\text{E}} = \mathcal{P}re(\text{E}) \bullet \left\{ \begin{array}{lcl} \text{RTL} & = & \text{ACTIF} \boxtimes S \\ \text{ACTIF}^{+} & = & (\text{SET} \boxplus \text{ACTIF}) \boxtimes \neg\text{RESET} \end{array} \right\}$$

Present

The circuit for `P`$present\ S\{\text{P}_1\}$`else`$\{\text{P}_2\}$ is described in figure 12 in appendix E. Let E be an input environment, the SET control wire is propagated to the then operand P_1 assuming signal S is present while it is propagated to the else operand P_2 assuming that S is absent. The resulting environment E' is the $\boxplus$ law applied to the respective outgoing environments of P_1 and P_2. Let E' be $\langle$`P`$present\ S\{\text{P}_1\}$`else`$\{\text{P}_2\}\rangle_{\text{E}}$, E' is defined as follows:

$$\mathrm{E}' = \left[\begin{array}{l} \langle P_1 \rangle_E \blacktriangleleft (S_{def}.S_{val})\boxplus \\ \langle P_2 \rangle_E \blacktriangleleft (S_{def}.\overline{S_{val}})\boxplus \\ \mathrm{E} \blacktriangleleft (\overline{S_{def}}.\overline{S_{val}})\boxplus \\ \mathrm{E}^{\top} \blacktriangleleft (\overline{S_{def}}.S_{val}) \end{array}\right] \bullet \left\{\begin{array}{lcl} \mathrm{SET_{P_1}} & = & \mathrm{SET} \blacktriangleleft (S_{def}.S_{val}) \\ \mathrm{SET_{P_2}} & = & \mathrm{SET} \blacktriangleleft (S_{def}.\overline{S_{val}}) \\ \mathrm{RESET_{P_1}} & = & \mathrm{RESET} \\ \mathrm{RESET_{P_2}} & = & \mathrm{RESET} \\ \mathrm{RTL} & = & \mathrm{RTL_{P_1}} \boxplus \mathrm{RTL_{P_2}} \boxplus (1 \blacktriangleleft \overline{S_{def}}.S_{val}) \end{array}\right\}$$

Parallel

Figure 13 in appendix E shows the circuit for $\mathrm{P}_1 \| \mathrm{P}_2$. The output environment contains the upper bound of respective events in the output environments of P_1 and P_2. The parallel is ready to leave when both P_1 and P_2 are:

$$\langle P_1 \| P_2 \rangle_E = \langle P_1 \rangle_E \boxplus \langle P_2 \rangle_E \bullet \left\{\begin{array}{lcl} \mathrm{SET_{P_1}} & = & \mathrm{SET} \\ \mathrm{SET_{P_2}} & = & \mathrm{SET} \\ \mathrm{RESET_{P_1}} & = & \mathrm{RESET} \\ \mathrm{RESET_{P_2}} & = & \mathrm{RESET} \\ \mathrm{ACTIF_1}^{+} & = & (\mathrm{RTL_{P_1}} \boxplus \mathrm{ACTIF_1}) \boxtimes \neg \mathrm{RESET} \\ \mathrm{ACTIF_2}^{+} & = & \mathrm{RTL_{P_2}} \boxplus \mathrm{ACTIF_2}) \boxtimes \neg \mathrm{RESET} \\ \mathrm{RTL} & = & (\mathrm{RTL_{P_1}} \boxplus \mathrm{ACTIF_1}) \boxtimes (\mathrm{RTL_{P_2}} \boxplus \mathrm{ACTIF_2}) \end{array}\right\}$$

Sequence

Figure 14 in appendix E shows the circuit for $\mathrm{P}_1 \gg \mathrm{P}_2$. The control is passed on from P_1 to P_2: when P_1 is ready to leave then P_2 get the control (equation 1) and P_1 is reseted (equation 2) :

$$\langle P_1 \gg P_2 \rangle_E = \langle P_1 \rangle_E \boxplus (\langle P_2 \rangle_{\langle P_1 \rangle_E} \blacktriangleleft \mathrm{RTL_{P_1}}_{val}) \bullet \left\{\begin{array}{lcl} \mathrm{SET_{P_1}} & = & \mathrm{SET} \\ \mathrm{SET_{P_2}} & = & \mathrm{RTL_{P_1}}(1) \\ \mathrm{RESET_{P_1}} & = & \mathrm{RESET} \boxplus \mathrm{RTL_{P_1}}(2) \\ \mathrm{RESET_{P_2}} & = & \mathrm{RESET} \\ \mathrm{RTL} & = & \mathrm{RTL_{P_2}} \end{array}\right\}$$

Abort

The `abort` statement has for semantic the circuit described in figure 15 in appendix E. A register is introduced since the operator semantic is to not react instantaneously to the presence of the aborting signal:

$$\langle \mathtt{a}\mathit{bort}\ \mathrm{P}\ \text{when}\ S \rangle_E = \langle \mathrm{P} \rangle_E \bullet \left\{\begin{array}{lcl} \mathrm{SET_P} & = & \mathrm{SET} \\ \mathrm{RESET_P} & = & (\neg \mathrm{RESET} \boxdot S) \boxplus \mathrm{RESET} \\ \mathrm{RTL} & = & (\neg S \boxdot \mathrm{RTL_P}) \boxplus (S \boxtimes (\mathrm{SET} \boxplus \mathrm{ACTIF})) \\ \mathrm{ACTIF}^{+} & = & (\mathrm{SET} \boxplus \mathrm{ACTIF}) \boxtimes \neg \mathrm{RESET} \end{array}\right\}$$

Loop

The statement `loop{P}` has for semantic the circuit described in figure 16 in appendix E. The `loop` statement does not terminate and similarly to its behavioral semantic, its circuit semantic is equal to the one of P $\gg$ loop P:

$$\langle \texttt{l}oop\ \mathrm{P}\rangle_E = \langle \mathrm{P}\rangle_E \bullet \left\{ \begin{array}{lcl} \mathrm{SET_P} & = & \mathrm{SET} \boxplus \mathrm{RTL_P} \\ \mathrm{RESET_P} & = & \mathrm{RESET} \\ \mathrm{RTL} & = & 0 \end{array} \right\}$$

Local

The `local` S {P} statement restricts the scope of S to sub statement P. At the opposite to interface signals, such a signal can be both tested and emitted. Thus, we consider that S is a new signal that does not belong to the input environment (it always possible, up to a renaming operation). Let SET, RESET and RTL be the respective input and output wires of the circuit, the equations of `local` S {P} are:

$$\langle \texttt{l}ocal\ \mathrm{S}\ \{\mathrm{P}\}\rangle_E = \langle \mathrm{P}\rangle_E \bullet \left\{ \begin{array}{lcl} \mathrm{SET_P} & = & \mathrm{SET} \\ \mathrm{RESET_P} & = & \mathrm{RESET} \\ \mathrm{RTL} & = & \mathrm{RTL_P} \\ S & = & \bot \end{array} \right\}$$

Run{P}

The circuit for `run` statement is described in figure 17 in appendix E. Intuitively, `run` {P} behaves similarly to P if P does not react instantaneously, and to `pause` $\parallel$ P. Thus, we get the following equation systems:

$$\langle \texttt{r}un\ \mathrm{P}\rangle_E = \mathcal{P}re(\mathrm{E}) \boxplus \langle \mathrm{P}\rangle_E \bullet \left\{ \begin{array}{lcl} \mathrm{SET_P} & = & \mathrm{SET} \\ \mathrm{RESET_P} & = & \mathrm{RESET} \\ \mathrm{ACTIF_1}^+ & = & (\mathrm{SET} \boxplus \mathrm{ACTIF_1}) \boxtimes \neg \mathrm{RESET} \\ \mathrm{ACTIF_2}^+ & = & (\mathrm{RTL_P} \boxplus \mathrm{ACTIF_2}) \boxtimes \neg \mathrm{RESET} \\ \mathrm{RTL} & = & \mathrm{ACTIF_1} \boxtimes (\mathrm{RTL_P} \boxplus \mathrm{ACTIF_2}) \end{array} \right\}$$

Automata

As already discussed, an automata is a finite set of *macro states*. A macro state does not react instantaneously, but takes at least an instant. Figure 18 in appendix E describes the circuit semantic for $\mathcal{A}(\mathcal{M}, \mathcal{T}, \mathcal{C}ond, M_f, \mathcal{O}, \lambda)$. The equational semantic of automata is the following set of equations:

$$\langle \mathcal{A} \rangle_E = \sum_{M \in \mathcal{M}} \langle M \rangle_{\mathcal{E}_M} \bullet \bigcup_{M \in \mathcal{M}} \left\{ \begin{array}{lcl} \mathrm{SET_M} & = & (\sum_{\mathrm{M}_i \to \mathrm{M} \in \mathcal{T}} \mathrm{RTL}_{\mathrm{M}_i} \blacktriangleleft c_{\mathrm{M}_i \to \mathrm{M}}) \boxplus (\mathrm{SET} \blacktriangleleft \sum_{\to \mathrm{M} \in \mathcal{T}} c_{\to \mathrm{M}}) \\ \mathrm{RESET_M} & = & \sum_{\mathrm{M} \to \mathrm{M}_i \in \mathcal{T}} \mathrm{RTL_M} \blacktriangleleft c_{\mathrm{M} \to \mathrm{M}_i}) \boxplus \mathrm{RESET} \\ \mathrm{RTL} & = & \mathrm{RTL}_{\mathrm{M_f}} \end{array} \right\}$$

where $\mathcal{E}_M$ is defined by:

$$E[s \leftarrow 1 \blacktriangleleft c_{\to M} \mid s \in \lambda(\to M)] \blacktriangleleft \sum_{\to M \in \mathcal{T}} c_{\to M} \boxplus \sum_{M_i \to M \in \mathcal{T}} (\langle M_i \rangle_{E[s \leftarrow 1 \blacktriangleleft c_{M_i \to M} \mid s \in \lambda(M_i \to M)]}) \blacktriangleleft c_{M_i \to M}$$

To complete automata circuit semantic definition, we now detail the circuit for macro states. Let M be a single macro state (which does not contain a `run` P instruction), then its associated circuit is similar to the one of `pause`:

$$\langle \mathrm{M} \rangle_{\mathrm{E}} = \mathcal{P}re(\mathrm{E}) \bullet \left\{ \begin{array}{lcl} \mathrm{ACTIF}^+ & = & (\mathrm{SET_M} \boxplus \mathrm{ACTIF}) \boxtimes \neg \mathrm{RESET_M} \\ \mathrm{RTL_M} & = & ACTIF \end{array} \right\}$$

Otherwise, if the macro state M contains a `run` P instruction, its circuit is the combination of equations for single macro state and equations for `run` operator:

$$\langle \mathrm{M} \rangle_{\mathrm{E}} = \mathcal{P}re(\langle \mathrm{P} \rangle_{\mathrm{E}}) \bullet \left\{ \begin{array}{lcl} \mathrm{SET_P} & = & \mathrm{ACTIF_1} \\ \mathrm{RESET_P} & = & \mathrm{RESET_M} \\ {\mathrm{ACTIF_1}}^+ & = & (\mathrm{SET_M} \boxplus \mathrm{ACTIF_1}) \boxtimes \neg \mathrm{RESET_M} \\ {\mathrm{ACTIF_2}}^+ & = & (\mathrm{RTL_P} \boxplus \mathrm{ACTIF_2}) \boxtimes \neg \mathrm{RESET_M} \\ \mathrm{RTL_M} & = & \mathrm{ACTIF_1} \boxtimes (\mathrm{RTL_P} \boxplus \mathrm{ACTIF_2}) \end{array} \right\}$$

Notice that a register is generated for each state, but in practice, we create only $\log_2 n$ registers if the automaton has n states according to the well-known binary encoding of states.

4.3 Equivalence between Behavioral and Circuit Semantic

The circuit semantic allows us to compile LE programs in a compositional way. Given a non basic statement p Op q (let Op be an operator of LE), then its associated circuit is deduced from $\langle p \rangle_E$ and $\langle q \rangle_E$ applying the semantic rules. On the other hand, the behavioral semantic gives a meaning to each program and is logically correct, and we prove now that these two semantic agree on both the set of emitted signals and the termination flag value for a LE program P. To prove this equivalence, we consider a global input environment E containing input events and output events set to $\perp$. Considering the circuits semantic, the global environment (denoted $E_{\mathcal{C}}$) is $E \cup P_S \cup P_R$.
To prove the equivalence between behavioral and circuit semantic, first we introduce a notation: let P be a LE statement, $SET(P)$, $RESET(P)$ and $RTL(P)$ will denote respectively the SET, RESET and RTL wires of P. Second, we introduce the notion of size for a statement.

Definition

We define $\lceil P \rceil$, the size of P as follows:

- $\lceil$`nothing`$\rceil$ = 1;
- $\lceil$ `halt`$\rceil$ = 1;
- $\lceil$`emit`$\rceil$ = 1;
- $\lceil$`pause`$\rceil$ = 1;
- $\lceil$`wait`$\rceil$ = 1;
- $\lceil$`present` $\{P_1\}$ `else` $\{P_2\}\rceil$ = $\lceil P_1 \rceil$ + $\lceil P_2 \rceil$ +1;
- $\lceil P_1 \| P_2 \rceil$ = $\lceil P_1 \rceil$ + $\lceil P_2 \rceil$ +1;
- $\lceil P_1 \gg P_2 \rceil$ = $\lceil P_1 \rceil$ + $\lceil P_2 \rceil$ +1;
- $\lceil$`abort` $\{P\}$ `when` $S\rceil$ = $\lceil P \rceil$ +1;
- $\lceil$`loop` $\{P\}\rceil$ = $\lceil P \rceil$ +1;
- $\lceil$`local`S $\{P\}\rceil$ = $\lceil P \rceil$ +1;
- $\lceil$`automata`$(\mathcal{M}, \mathcal{T})\rceil$ = $\sum \lceil M_i \rceil$ such that $M_i \in \mathcal{M}$ +1;

Theorem. *Let P be a* LE *statement and E_C an input environment, For each reaction, the following property holds:*
$\Gamma(P) \xrightarrow[E]{E', \mathrm{TERM}(P)_{val}} \Gamma(P)'$, *where* $E = E_C - \{w | w \in P_S \text{ or } w \in P_R\}$; $TERM = RTL(P)_{val}$; *and* $\langle P \rangle_{E_C} \restriction_O = E' \restriction_O$

Proof

We perform an inductive proof on the size of P. Notice that the proof requires to distinguish the initial reaction from the others. In this reaction, $SET(P) = 1$ and it is the only instant when this equality holds. For statement reacting instantaneously, we consider only an initial reaction since considering following reactions is meaningless for them.

$\lceil P \rceil$ = **1**

, We perform a proof by induction on the length of P. First, we prove the theorem for basic statements whose length is 1. According to the previous definition of $\lceil\rceil$, P is either `nothing`, `halt`, `emit`, `pause` or `wait`.

1. $P =$ `nothing`;

   then $\Gamma(P)$ = `nothing`. Following the equational semantic for `nothing` statement:

   $$\langle P \rangle_{E_\mathcal{C}} = E_\mathcal{C} \bullet \{RTL(P) = SET(P)\}$$

   Hence, $\langle P \rangle_{E_\mathcal{C}} \restriction_O = E_\mathcal{C} \restriction_O = E \restriction_O = E' \restriction_O$. Moreover, $RTL(P) = SET(P) = 1$ thus $RTL(P)_{val} = 1$;

2. P = `halt`;

   then $\Gamma(P)$ = `halt`. Similarly to `nothing`, $\langle P \rangle_{E_\mathcal{C}} \restriction_O = E' \restriction_O$ and $RTL(P) = 0$ thus $RTL(P)_{val}$ = 0;

3. P = `emit` S;

   then $\Gamma(P) = !S$. As well in the behavioral rule for `!` as in the circuit equations for `emit`, we set the status of signal S to 1 in the respective environments. From the definition, $E_\mathcal{C} \restriction_O = E \restriction_O$ thus obviously, $\langle P \rangle_{E_\mathcal{C}} \restriction_O = E' \restriction_O$. Moreover, $RTL(P) = SET(P) = 1$ thus, $RTL(P)_{val} = 1$.

4. P = `wait` S;

   According to the circuit semantic, $\mathcal{C}(P)$ has a register wire and we denote it $ACTIF(P)$. The equations for `wait` are:

   $$\langle P \rangle_E = \mathcal{P}re(\mathrm{E}) \bullet \left\{ \begin{array}{lcl} RTL(P) & = & ACTIF(P) \boxtimes S \\ ACTIF(P)^+ & = & (SET(P) \boxplus ACTIF(P)) \boxtimes \neg RESET(P) \end{array} \right\}$$

   The proof of the theorem falls into two cases:

   (a) ACTIF(P)=0, we are in the initial reaction and then $SET(P) = 1$, $RESET(P) = 0$. it is obvious that $ACTIF(P)^+ = 1$. Then $ACTIF(P)$ becomes 1 in the environment according to the $\mathcal{P}re$ operation and all output wires keep their status in $E'_\mathcal{C}$. When such a reaction occurs, in the behavioral semantic definition, the $wait$ rule is applied. Following this rule $E' = E$. Thus, $\langle P \rangle_{E_\mathcal{C}} \restriction_O = E_\mathcal{C} \restriction_O = E \restriction_O = E' \restriction_O$, according to the $\mathcal{P}re$ operation definition which does not concern output signals. From the equations above, we get $RTL(P) = 0$ whatever the status of S is and then $RTL(P)_{val} = 0$; this is in compliance with the $wait$ rule. Another situation where $ACTIF(P) = 0$ is when $RESET(P)$ has been set to 1 in the previous reaction. This case occurs only if the wait statement is the first part of a $\gg$ operator or the internal statement of an `abort` operator. In both cases, $RTL(P) = 0$ then $RTL(P)_{val} = 0 = TERM_{\Gamma(P)}$ and in both semantic the outgoing environments remain unchanged and then the theorem still holds.

(b) ACTIF(P) = 1. we are not in the initial reaction. Then, the corresponding rules applied in behavioral semantic are either $iwait1$ or $iwait2$ depending of S status in the environment. Similarly to item 1, neither $iwait1$ and $iwait2$ rules nor $\mathcal{P}re$ operation change environment output signals, thus $\langle P\rangle_{E_\mathcal{C}}\restriction_O = E'\restriction_O$.

If $S^1 \in E_\mathcal{C}$ then $S^1 \in E$ since it is either an input signal or a local one for the statement and then we apply rule $iwait1$, then $RTL(P) = 1$ and thus $RTL(P)_{val} = 1$. Otherwise, if $S^x \in E_\mathcal{C},\ x \neq 1$ and $ACTIF(P) \boxtimes S = (1, S_{def}.S_{val}.ACTIF(P)_{val}) = (1, 0)$ for $S = 0, \perp$ or $\top$. Thus $RTL(P) = 0$ and $RTL(P)_{val} = 0$.

$\lceil P \rceil$ = **n**

Now we study the inductive step, Assume that the theorem holds for statement whose length is less than n. We study the case where the size of P is n. Then P is either `present`, $\|$, $\gg$, `abort`, `loop`, `local` or `automata` statement.

1. $P =$ `present` $S\,\{P_1\}$ `else` $\{P_2\}$;

   Thus, according to the equational semantic, we know that:

$$\left\{\begin{array}{l}\langle P_1\rangle_{E_\mathcal{C}} \blacktriangleleft (S_{def}.S_{val}) \boxplus \\ \langle P_2\rangle_{E_\mathcal{C}} \blacktriangleleft (S_{def}.\overline{S_{val}}) \boxplus \\ E_\mathcal{C} \blacktriangleleft (\overline{S_{def}}.\overline{S_{val}}) \boxplus \\ E_{\mathcal{C}\top} \blacktriangleleft (\overline{S_{def}}.S_{val})\end{array}\right\} \bullet \{RTL(P) = RTL(P_1) \boxplus RTL(P_2) \boxplus (1 \blacktriangleleft \overline{S_{def}}.S_{val})\} \subset \langle P\rangle_{E_\mathcal{C}}$$

   On the other hand, $\Gamma(P) = S\ ?p_1 : p_2$ where $p_1 = \Gamma(P_1)$ and $p_2 = \Gamma(P_2)$. The behavioral semantic relies on the four rules defined in section 3.2:

   By induction , we know that $\langle P_1\rangle_{E_\mathcal{C}}\restriction_O = E'_1\restriction_O$ and $\langle P_2\rangle_{E_\mathcal{C}}\restriction_O = E'_2\restriction_O$ where E'_1 (resp E'_2) is the output environment of p_1 (resp p_2) computed from E input environment, and $RTL(P_1)_{val} = TERM_{p_1}$ and $RTL(P_2)_{val} = TERM_{p_2}$. To prove the theorem for present operator, we study the different possible status of S in the input environment (common to both semantic).

   (a) If S is present, then $S_{def} = 1$ and $S_{val} = 1$. For the output signal valuation, since $S_{def}.S_{val} = 1$, from the induction hypothesis we deduce that $\langle P\rangle_{E_\mathcal{C}}\restriction_O = E'\restriction_O$. Concerning the RTL wire and termination flag, if we consider present operator equations, since $S_{def} = 1$ and $S_{val} = 1$, we deduce that $SET(P_1) = 1$ and $SET(P_2) = 0$. Thus $RTL(P_2) = 0$ too: either P_2 has no register and then its RTL value depends straightly of the SET value, or P_2 has a register. In this case, its RTL value depends of register value, but this latter cannot be 1 while the SET value is 0. Thus, $RTL(P)_{val} = RTL(P_1)_{val} = TERM_{P_1} = TERM_{\Gamma(P)}$ with respect to rule $present0$ in the behavioral semantic.

   (b) If S is absent, the prove is similar with $S_{def} = 1$ and $S_{val} = 0$ and according to the fact that rule $present1$ is applied from the behavioral semantic.

(c) If S status is $\perp$, then $S_{def} = 0$ and $S_{val} = 0$. In this case $\langle P \rangle_{E_C} = E_C$ and $E' = E$, thus the result concerning outputs is obvious by induction. Concerning RTL wires and termination flag, since $S_{def} = 0$ thus both $SET(P_1)$ and $SET(P_2)$ are 0 and then also $RTL(P_1)$ and $RTL(P_2)$ are. Thus $RTL(P) = 0$ and $RTL(P)_{val} = 0 = TERM_{\Gamma(P)}$ according to rule $present3$ from behavioral semantic.

(d) If S has status $\top$, then an error occurs and in both semantic all signals in the environment are set to $\top$. In this case, $RTL(P) = 1$ and according to rule $present4$ of behavioral semantic, $RTL(P)_{val} = TERM_{\Gamma(P)} = 1$.

2. $P = P_1 \| P_2$;

Thus, equations for P are the following:

$$\langle P \rangle_{E_C} = \langle P_1 \rangle_{E_C} \boxplus \langle P_2 \rangle_{E_C} \bullet \left\{ \begin{array}{lcl} SET(P_1) & = & SET(P) \\ SET(P_2) & = & SET(P) \\ RESET(P_1) & = & RESET(P) \\ RESET(P_2) & = & RESET(P) \\ ACTIF_1(P)^+ & = & (RTL(P_1) \boxplus ACTIF_1(P)) \boxtimes \neg RESET(P) \\ ACTIF_2(P)^+ & = & RTL(P_2) \boxplus ACTIF_2(P) \boxtimes \neg RESET(P) \\ RTL(P) & = & (RTL(P_1) \boxplus ACTIF_1(P)) \boxtimes \\ & & (RTL(P_2) \boxplus ACTIF_2(P)) \end{array} \right\}$$

In PLE process algebra, $\Gamma(P) = p_1 \| p_2$, where $p_1 = \Gamma(P_1)$ and $p_2 = \Gamma(P_2)$. We recall the $parallel$ rule of behavioral semantic for $\|$:

$$\frac{p_1 \xrightarrow[E]{E_1', TERM_{p_1}} p_1' \quad , \quad p_2 \xrightarrow[E]{E_2', TERM_{p_2}} p_2'}{p \xrightarrow[E]{E_1' \boxplus E_2', TERM_{p_1}.TERM_{p_2}} p_1' \| p_2'}$$

By induction , we know that $\langle P_1 \rangle_{E_C} \restriction_O = E_1' \restriction_O$ and $\langle P_2 \rangle_{E_C} \restriction_O = E_2' \restriction_O$ and $RTL(P_1)_{val} = TERM_{p_1}$ and $RTL(P_2)_{val} = TERM_{p_2}$.

Both equational and behavioral semantic perform the same $\boxplus$ operation on the environments resulting of the computation of the respective semantic on the two operands. Thus, the result concerning the outputs is straightly deduced from the induction hypothesis.

Concerning the RTL wire, $(RTL(P_1) \boxtimes RTL(P_2))_{val} = (RTL(P_1)_{val}.RTL(P_2)_{val}$ by definition of $\boxtimes$ operation and according to the fact that $RTL(P_1)_{def} = RTL(P_2)_{def} = 1$,and by induction $RTL(P)_{val} = TERM_{p_1}.TERM_{p_2} = TERM_{\Gamma(P)}$.

3. $P = P_1 \gg P_2$;

The equations for $\gg$ operator are the following:

$$\langle P \rangle_{E_C} = \langle P_1 \rangle_{E_C} \boxplus (\langle P_2 \rangle_{\langle P_1 \rangle_{E_C}} \blacktriangleleft RTL(P_1)) \bullet \left\{ \begin{array}{lcl} SET(P_1) & = & SET(P) \\ SET(P_2) & = & RTL(P_1)(1) \\ RESET(P_1) & = & RESET(P) \boxplus RTL(P_1)(2) \\ RESET(P_2) & = & RESET(P) \\ RTL(P) & = & RTL(P_2) \end{array} \right\}$$

In PLE process algebra, $\Gamma(P) = p_1 \gg p_2$ where $p_1 = \Gamma(P_1)$ and $p_2 = \Gamma(P_2)$.

The proof depends of the value of $RTL(P_1)$ in the equational semantic:

(a) $RTL(P_1) = 0$;

By induction we know that $p_1 \xrightarrow[E]{E'_1, TERM_{p_1}} p'_1$ and $TERM_{p_1} = RTL(P_1)_{val} = 0$. Then, in the behavioral semantic , rule $sequence1$ is applied. Thus, $TERM_{\Gamma(P)} = 0$ and $E' = E'_{p_1}$. In the equational semantic, $SET(P_2) = RTL(P_1)$ thus $SET(P_2) = 0$ and so is $RTL(P_2)$ (see the proof of `present` operator) and $RTL(P)$ too. $RTL(P_1)_{val} = 0$ and according to $\blacktriangleleft$ definition, $(\langle P_2 \rangle_{\langle P_1 \rangle_{E_\mathcal{C}}} \blacktriangleleft RTL(P_1)) = E_\perp$. Thus, $\langle P \rangle_{E_\mathcal{C}} = \langle P_1 \rangle_{E_\mathcal{C}} \bullet \mathcal{C}(P)$, and $\langle P \rangle_{E_\mathcal{C}} \restriction_0 = \langle P_1 \rangle_{E_\mathcal{C}} \restriction_O$. On the other hand, in behavioral semantic, we have $E' \restriction_O = E'_{p_1}$. Thus, from induction hypothesis, we deduce that: $\langle P \rangle_{E_\mathcal{C}} \restriction_O = E' \restriction_O$.

(b) $RTL(P_1) = 1$;

In this case, $TERM_{p_1} = RTL(P_1)_{val} = 1$ and rule $sequence2$ is applied in the behavioral semantic. By induction, we know that $TERM_{p_2} = RTL(P_2)_{val}$. But, $RTL(P) = RTL(P_2)$ then $RTL(P)_{val} = TERM_p$. For environments, By induction, we also know that $\langle P_1 \rangle_{E_\mathcal{C}} \restriction_O = E'_{p_1} \restriction_O$. In both semantic, the only way to change the value of an output signal in the environment is with the help of the emit operator. Then, if the status of an output signal o change in $\langle P_1 \rangle_{E_\mathcal{C}}$ it is because P_2 involves an emit o instruction. Hence, relying on the induction hypothesis, we know that o has the same status in $\langle P_1 \rangle_{E_\mathcal{C}}$ and in E'_{p_1}. But, o status cannot be changed in two different ways in $\langle P \rangle_{E_\mathcal{C}}$ and E' since emit operator performs the same operation on environments in both semantic.

4. P = `abort` P_1 `when` S;

Thus, the output environment is the solution of the following equations:

$$\langle P \rangle_{E_\mathcal{C}} = \langle P_1 \rangle_{E_\mathcal{C}} \bullet \left\{ \begin{array}{lcl} SET(P_1) & = & SET(P) \\ RESET(P_1) & = & (\neg(RESET(P) \boxdot S) \boxplus RESET(P) \\ RTL(P) & = & (\neg S \boxdot RTL(P_1)) \boxplus (S \boxtimes (SET(P) \boxplus ACTIF(P))) \\ ACTIF(P)^+ & = & (SET(P) \boxplus ACTIF(P)) \boxtimes \neg RESET(P) \end{array} \right\}$$

$\Gamma(P) = p_1 \uparrow s$ where $p_1 = \Gamma(P_1)$.

First, notice that in $abort1$, $abort2$ and $abort3$ of behavioral semantic, the output environment E' is E'_{p_1}. Similarly in the equational semantic $E'_\mathcal{C}$ is $E'_{\mathcal{C}_1}$ improved by the set of connexion wire equations for abort statement. Then, applying the induction hypothesis, we can deduce $E' \restriction_O = \langle P \rangle_{E_\mathcal{C}} \restriction_0$.

Now, we prove that termination wire coincides with termination flag in respective equational and behavioral semantic. We study first the case where we S is present and then the case when it is not.

(a) $S^1 \in E$;

Thus, $S^1 \in E_{\mathcal{C}}$ too. In this case, $RTL(P) = SET(P) \boxplus ACTIF(P)$. In the initial reaction $SET(P) = 1$ and $ACTIF(P) = 0$ and in further reaction $SET(P) = 0$ and $ACTIF(P) = 1$. Then, in all reactions $RTL(P) = 1$. On the other hand, it is rule $abort1$ that is applied in behavioral semantic and thus $RTL_p = 1$. Hence, $RTL(P)_{val} = TERM_p$. However, $ACTIF(P)$ can become 0. But, that means that in a previous reaction $RESET(P) = 1$ and P is encompassed in a more general statement P_g which is either another abort or a sequence statement since there are the only operators that set the $RESET$ wire to 1. If P_g is an abort statement, its abortion signal is 1 in the input environment and then we are in one of the previous case already studied. Otherwise, that means that P is encompassed in the first operand of P_g whose RTL is 1 and we can rely on the reasoning performed for sequence operator to get the result we want.

(b) $S^1 \notin E$; Thus, $S^1 \notin E_{\mathcal{C}}$ too. If we expand the value of S in the RTL equation, we get $RTL(P) = RTL(P_1)$. In the behavioral semantic either rule $abort1$ or $abort2$ is applied according to the value of $TERM_{p_1}$. But, whatever this value is, by induction we get the result.

5. P = `loop` { P_1 };

Thus,

$$\langle P \rangle_{E_{\mathcal{C}}} = \langle P_1 \rangle_{E_{\mathcal{C}}} \bullet \left\{ \begin{array}{lcl} SET(P_1) & = & SET(P) \boxplus RTL(P_1) \\ RESET(P_1) & = & RESET(P) \\ RTL(P) & = & 0 \end{array} \right\}$$

$\Gamma(P) = \Gamma(P_1)*$ and rule *loop* is applied in the behavioral semantic to compute the reaction of $\Gamma(P)$. According to this latter, $p_1* \xrightarrow[E]{E'_1,0} p'_1 \gg p_1*$ when $p_1 \xrightarrow[E]{E'_1,TERM_{p_1}} p'_1$. By induction, we know that $\langle P_1 \rangle_{E_{\mathcal{C}}} \restriction_O = E'_{p_1} \restriction_O$ thus $\langle P \rangle_{E_{\mathcal{C}}} = E' \restriction_O$ and $RTL(P) = 0$ thus $RTL(P)_{val} = 0 = TERM_{\Gamma(P)}$.

6. P = `local` S {P_1};

According to the equational semantic, the following equations defined the local operator:

$$\langle P \rangle_{E_{\mathcal{C}}} = \langle P_1 \rangle_{E_{\mathcal{C}}} \bullet \left\{ \begin{array}{lcl} SET(P_1) & = & SET(P) \\ RESET(P_1) & = & RESET(P) \\ RTL(P) & = & RTL(P_1) \\ S & = & \bot \end{array} \right\}$$

In PLE process algebra $\Gamma(P) = p_1 \backslash S$ where $\Gamma(P_1) = p_1$. *local* rule is applied in the behavioral semantic: $\Gamma(P) \xrightarrow[E]{E'_1 - \{S\},TERM_{p_1}} p'_1 \backslash S$ when $p_1 \xrightarrow[E \cup \{S\}]{E'_1,TERM_{p_1}} p'_1$. Following the induction hypothesis, $\langle P_1 \rangle_{E_{\mathcal{C}}} \restriction_O = E'_{p_1} \restriction_O$ and $RTL(P_1)_{val} = TERM_{p_1}$.

Then $\langle P_1 \rangle_{E_{\mathcal{C}}} - \{S\} \restriction_O = E'_{p_1} \restriction_O - \{S\}$ and $\langle P \rangle_{E_{\mathcal{C}}} \restriction_O = E' \restriction_O$, straightly from the induction hypothesis.

7. P = `run` $\{P_1\}$

   The `run` operator is not a primitive one, and we defined it as: wait `tick` $\gg P_1$. Thus, the property holds for `run` operator since it holds for both wait and $\|$ operators.

8. $P = \mathcal{A}(\mathcal{M}, \mathcal{T}, \mathcal{C}ond, M_f, \mathcal{O}, \lambda)$.

   Automata are both terms in PLE process algebra and programs in LE language. The equations for automata are the following:

$$\langle P \rangle_{E_\mathcal{C}} = \sum_{M \in \mathcal{M}} \langle M \rangle_{\mathcal{E}_M} \quad \bullet \bigcup_{M \in \mathcal{M}} \left\{ \begin{array}{lcl} SET(M) & = & (\sum_{M_i \rightarrow M \in \mathcal{T}} RTL(M_i) \blacktriangleleft c_{M_i \rightarrow M}) \quad \boxplus \\ & & (SET(P) \blacktriangleleft \sum_{\rightarrow M \in \mathcal{T}} c_{\rightarrow M}) \\ RESET(M) & = & \sum_{M \rightarrow M_i \in \mathcal{T}} RTL(M) \blacktriangleleft c_{M \rightarrow M_i} \boxplus \\ & & RESET(P) \\ RTL(P) & = & RTL(M_f) \end{array} \right\}$$

   where $\mathcal{E}_M$ is defined by:

$$E[s \leftarrow 1 \blacktriangleleft c_{\rightarrow M} \mid s \in \lambda(\rightarrow M)] \blacktriangleleft \sum_{\rightarrow M \in \mathcal{T}} c_{\rightarrow M}) \boxplus$$
$$\sum_{M_i \rightarrow M \in \mathcal{T}} (\langle M_i \rangle_{E[s \leftarrow 1 \blacktriangleleft c_{M_i \rightarrow M} \mid s \in \lambda(M_i \rightarrow M)]}) \blacktriangleleft c_{M_i \rightarrow M}$$

   First of all, let us consider macro states. These latter are either single macro states equivalent to a `pause` statement, or they contains a `run` P instruction and then are equivalent to a `pause` $\gg$ P instruction. In both cases, we have already prove that the theorem holds. Now, to prove the theorem for automata, we perform an inductive reasoning on the sequence of reactions.

   In the first reaction $SET(P) = 1$. All the $RTL(M)$ are 0, since macro states have at least a one instant duration, thus $RTL(P)_{val} = 0$. On the other side, in behavioral semantic, rule $automaton0$ is applied and $TERM_p = 0$ too.

   For environments, for each macro states M, in the first reaction

$$\mathcal{E}_M = E_\mathcal{C}[s \leftarrow 1 \text{ when } c_{\rightarrow M} = 1 \text{ and } s \in \lambda(\rightarrow M) \text{ and } \rightarrow M \in \mathcal{T}]$$

   When looking at equations related to M, we see that no output signal status can be modified in the first reaction: either it is a single macro state and then no output signal is modified whatever the reaction is, or it contains a `run` P_0 statement but $SET(P_0)$ cannot be true in the initial reaction and so no output signal status can't be modified (the only operator that modified output status is the `emit` one, and if the SET wire of an `emit` statement is not 1, the status of the emitted signal remains unchanged). Thus, as well the behavioral semantic in rule $automata0$ as the equational semantic set to 1 in their respective environment output signal emitted in the initial transitions that teach M. Hence, $E_\mathcal{C}{\restriction_O} = E{\restriction_O}$.

Now, we consider that the result is proved for the previous n reactions, and we prove the result for the $n+1$ reaction. In this reaction, for each macro state $M \neq M_f$, if there is no transition $M_i \rightarrow M$ such that $c_{M_i \rightarrow M}$ =1, then $\mathcal{E}_M = E_{\mathcal{C}_n}$ where $E_{\mathcal{C}_n}$ in the environment obtained after the previous n reactions. Thus, $\langle M \rangle_{\mathcal{E}_M}$ is $E_{\mathcal{C}_n} \bullet \mathcal{C}(M)$ where $\mathcal{C}(M)$ is the set of equations related to macro state M. In the behavioral semantic, it is rule $automaton1$ which is applied and thus relying both on induction hypothesis ensuring that $E_{\mathcal{C}_n} \restriction_O = E_n \restriction_O$ and on the fact that the theorem is true for macro state, we deduce the result. Concerning the $TERM$ flag, as M is not final, $SET(M_f) = 0$ and so is $RTL(M_f)$. Hence, $RTL(P) = 0$, thus $RTL(P)_{val} = 0$ too and $RTL_{\Gamma(P)}$ too (cf rule $automaton1$). On the other hand, if there is a transition $M_i \rightarrow M$ such that $c_{M_i \rightarrow M} = 1$, $\mathcal{E}_M = E_{\mathcal{C}_n} \boxplus \langle M_i \rangle_{E_{\mathcal{C}_n}[s \leftarrow 1 \ | s \in \lambda(M_i \rightarrow M)]}$. If there is a transition $M_k \rightarrow M$ such that $c_{M_k \rightarrow M}$ =1, thus similarly to the case where n = 1, we have

$$\mathcal{E}_M = E_{\mathcal{C}_n}[s \leftarrow 1 \; when \; c_{M_k \rightarrow M} = 1 \text{ and } s \in \lambda(M_k \rightarrow M) \text{ and } \rightarrow M \in \mathcal{T}]$$

Since $E_{\mathcal{C}_n}$ is the resulting environment of the previous instant, we know that $E_{\mathcal{C}_n} \restriction_O = E_n \restriction_O$ On the other hand, it is rule $automaton2$ that is applied in the behavioral semantic and the output environment is modified in the same way for both semantic. Similarly to the first instant, equations for M cannot modified the environment the first instant where SET_M is 1. Then, $E_{\mathcal{C}_{n+1}} \restriction_O = E_{n+1} \restriction_O$. In this case $RTL(M_f)$ is still 0 , thus $RTL(P) = 0$ and so is $RTL(P)_{val}$. Hence, according to rule $automaton2$, result for $TERM$ flag holds.

Now, we will consider that there is a transition $M_k \rightarrow M_f$ such that $c_{M_k \rightarrow M_f} = 1$ [5]. In this case, a similar reasoning to the case where M is not a final macrostate concerning output environments holds. For termination flag, in equational semantic, $RTL(P) = 1$ when $RTL(M_f) = 1$. A similar situation holds for behavioral semantic where it is rule $automata4$ which is applied. Then, the result is deduced from the general induction hypothesis since the size of macro states is less than the size of automata from the definition.

♣

5 LE Modular Compilation

5.1 Introduction

In the previous section, we have shown that every construct of the language has a semantic expressed as a set of ξ equations. The first compilation step is the generation of a ξ equation system for each LE program, According to the semantic laws described in section 4. Then, we translate each ξ circuit into a boolean circuit relying on the bijective map from ξ-algebra to $\mathbb{B} \times \mathbb{B}$ defined in section 3. This encoding allows us to translate ξ equation system into a boolean equation system (each equation being encoded by two boolean equations). Thus, we can rely on a constructive propagation law to implement equation system evaluation and then, generate code, simulate or link with external code. But this approach requires to find an evaluation order, valid for all synchronous instants. Usually,

[5] the demonstration is the same when the transition is initial (i.e $c \rightarrow M_f = 1$)

in the most popular synchronous languages existing, this order is static. This static ordering forbids any separate compilation mechanism as it is illustrated in the following example. Let us consider the two modules `first` and `second` compiled in a separate way. Depending on the order chosen for sorting independant variables of each modules, their parallel combination may lead to a causality problem (i.e there is a dependency cycle in the resulting equation system).

```
module first:
Input: I1,I2;
Output: O1,O2;
loop {
 pause >>
 {
 present I1 {emit O1}
 ||
 present I2 {emit O2}
 }
end
```

```
module second:
Input: I3;
Output: O3;
loop {
 pause >>  present I3 {emit O3}
}
end
```

```
module final:
Input: I;
Output O;
local  L1,L2 {
 run first[ L2\I1,O\O1,I\I2,L1\O2]
 ||
 run second[ L1\I3,L2\O3]
}
end
```

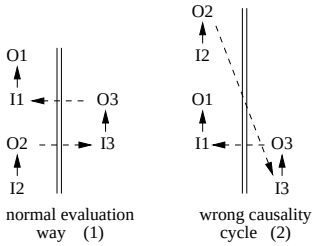


Figure 2: Causality cycle generation: O1, O2 and O3 signals are independent. But, when choosing a total order, we can introduce a causality cycle. If ordering (1) is chosen, in module final, taking into account the renaming, we obtain the system: { L1 = I, L2 = L1, O = L2 } which is well sorted. At the opposite, if we choose ordering (2), in module final we get: { L2 = L1, O = L2,L1 = I } which has a causality cycle.

Figure 2 describes a LE module calling two sub modules. Two compilation scenarios are shown on the right part of the figure. The first one leads to a sorted equation system while the second introduces a fake causality cycle that prevents any code generation. Independent signals must stay not related: we aim at building an incremental partial order. Hence, while ordering the equation system, we keep enough information on signal causality to preserve the independence of signals. At this aim, we define two variables for each equation, namely (*Early Date, Late Date*) to record the level when the equation *can* (resp. *must*) be evaluated. Each level is composed of a set of independent equations. Level 0 characterizes the equations evaluated first because they only depend of free variables, while level n+1 characterizes the equation needed the evaluation of variables from lower levels (from n to 0) to be evaluated. Equations of same level are independent and so can be evaluated whatever the chosen order is. This methodology is derived from the PERT method. This latter is well known for decades in the industrial production. Historically, this method has been invented for the spatial conquest, back to the 60th when the NASA was facing the problem of

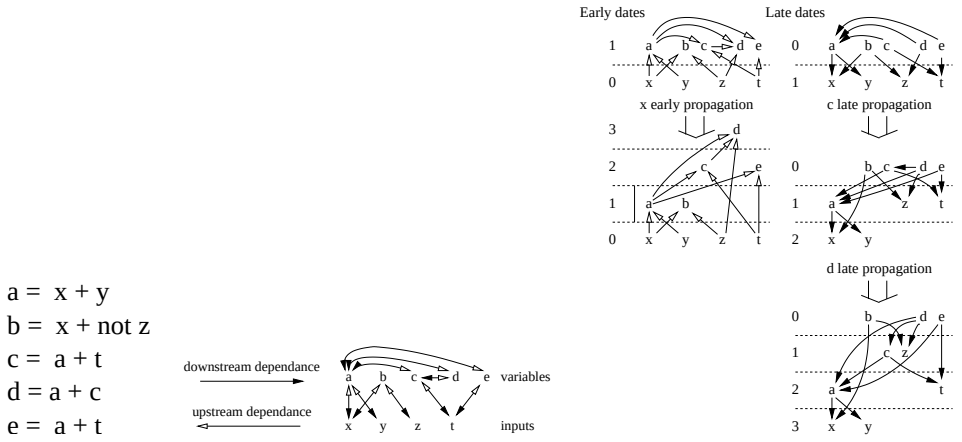
synchronizing 30,000 independent, thus "concurrent", dealers to built the Saturne V rocket.

5.2 Sort algorithm: a PERT family method

Usually, the PERT method is applied in a task management context and each task has a duration. In our usage, taking account duration of task makes no sense and the algorithm we rely on to implement the PERT method is simplified. It is divided into two phasis. The first step constructs a forest where each tree represents variable dependencies. Thus an initial partial order is built. The second step is the recursive propagation of early and late dates. If during the propagation, a cycle is found there is a causality cycle in the program. Of course the propagation ends since the number of variables is finite. At worst, if the algorithm is successful (no causality cycle is found), we can find a total order with a single variable per level (n variables and n levels).

5.2.1 Sorting algorithm Description

More precisely, the first step builds two dependency sets (*upstream*, *downstream*) for each variable with respect to the equation which defines it. This first algorithm is detailed in appendix B.1. The *upstream* set of a variable X is the set of variables needed by X to be computed while the *downstream* set is the variables that need the value of X to be evaluated. In practice, boolean equation systems are implemented using binary decision diagrams (BDDs). Consequently the computation of the downstream table is given for free by the BDD library.



(a) Equation system

(b) Dependences forest

(c) Date propagation

Figure 3: The dependence forest and propagation law application for a specific equation system.The different pert levels are specified on the left hand side of figure 3(c).

We illustrate, the sorting algorithm we built on an example. Let us consider the set of equations expressed in figure 3(a). After the first step, we obtain the dependencies forest described in figure 3(b), Then, we perform early and late dates propagation. Initially, all variables are considered independent and their dates (*early , late*) are set to (0,0). The second step recursively propagates the *Early* Dates from the input and the register variables to the output variables and propagates the *Late* Dates from the output variables to the input and the register ones according to a *n log n* propagation algorithm. The algorithm that implements this second phasis is detailed in appendix B.2. Following the example presented in figure 3(a), the algorithm results in the dependencies described in figure 3(c).

5.2.2 Linking two Partial Orders

The approach allows an efficient merge of two already sorted equation systems, useful to perform separate compilation. To link the forest computed for module 1 with the forest computed for module 2, we don't need to launch again the sorting algorithm from its initial step. In fact, it is sufficient to only adjust the $early(late)\ dates$ of the common variables to both equation systems and their dependencies. Notice that the linking operation applies ξ-algebra plus operator to merge common equations (i.e equations which compute the same variable). Then, we need to adjust evaluation dates: every output variable of module 1 propagates new $late\ date$ for every downstream variables. Conversely, every input variable of module 2 propagates new $early\ date$ for every upstream variables.

5.3 Practical Issues

We have mainly detailed the theoretical aspect of our approach, and in this section we will discuss the practical issues we have implemented.

5.3.1 Effective compilation

Relying on the equational semantic, we compile a LE program into a ξ-algebra equation system. We call the compilation tool that achieves such a task CLEM (Compilation of LE Module). In order to perform separate compilation of LE programs, we define an internal compilation format called LEC (LE Compiled code). This format is highly inspired from the Berkeley Logic Interchange Format (BLIF [6]). This latter is a very compact format to represent netlists and we just add to it syntactic means to record the *early date* and *late date* of each equation. Practically, CLEM compiler, among other output codes, generates LEC format in order to reuse already compiled code in an efficient way, thanks to the PERT algorithm we implement.

5.3.2 Effective Finalization

Our approach to compile LE programs into a sorted ξ equation system in an efficient way requires to be completed by what we call a *finalization* phasis to be effective. To generate code for simulation, verification or evaluation, we must start from a *valid* boolean equation systems, i.e we consider only equation systems where no event has value $\top$, since that means there is an error an we propagate

[6] `http://embedded.eecs.berkeley.edu/Research/vis`

this value to each element of the environment in the semantic previously described. Validity also means well sorted equation systems, to avoid to deal with programs having causality cycle. But in our approach we never set input event status to $absent$. Hence, we introduce a ***finalization*** operation which replaces all $\perp$ input events by $absent$ events and propagates this information in all equations related to local variables and outputs. Notice that the finalization operation is harmless. The sorting algorithm relies on propagation of signal status, and the substitution of $\perp$ by $absent$ cannot change the resulting sorted environment.
Let us illustrate the finalization mechanism on an example. In the following code $O1$ and $O2$ depends on the I status:

```
loop {
  present I {emit O1} else {emit O2}
  >> pause
}
```

Before finalization, we get the following equation system:
$O1_{def} = I_{def}$
$O1_{val} = I_{val}.I_{def}$
$O2_{def} = I_{def}$
$O2_{val} = \neg I_{val}.I_{def}$
We can see that $O1_{def}$ and $O2_{def}$ are not constant because I is not necessarily defined for each instant (i.e I_{def} can be 0 if I is $\perp$). After finalization I_{def} is set to 1 and I_{val} remains free. According to the mapping from ξ algebra to $\mathbb{B} \times \mathbb{B}$, an event X such that $X_{def} = 0$ is either $\top$ or $\perp$. Since, we discard equation systems where an event has value $\top$, To switch from $\perp$ value to $absent$ value, it is sufficient to set the def part of a variable to 1. Now for each logical instant the status (***present***, ***absent***) of I is known. The $O1$ and $O2$ equations become:
$O1_{def} = 1$
$O1_{val} = I_{val}$
$O2_{def} = 1$
$O2_{val} = \neg I_{val}$
We bring together compilation and finalization processus in a tool named CLEF(Compilation of LE programs and Finalization).

5.3.3 Compilation scheme

Now, we detail the toolkit we have to specify, compile , simulate and execute LE programs. A LE file can be directly written. In the case of automaton, it can be generated by automaton editor like `galaxy` too. Each LE module is compiled in a LEC file and includes one instance of the RUN module references. These references can be already compiled in the past by a first call of the `clem` compiler. When the compiled process will done, the finalization will simplify the final equations and generate a file in the target use: simulation, safety proofs, hardware description or software code. That is summed up in the figure 4.

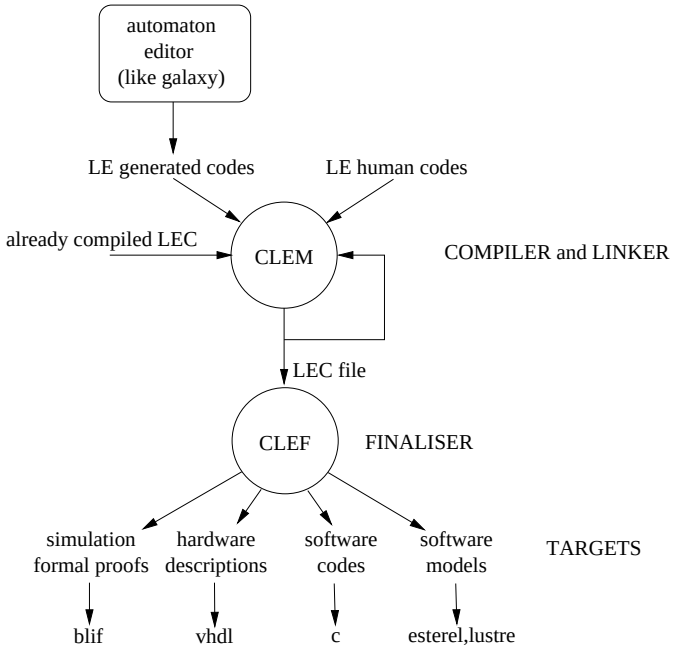


Figure 4: Compilation Scheme

5.4 Benchmark

To complement the experimentation of the example, we have done some tests about the CLEM compiler. So we are interested in the evolution of the generated code enlarging with respect to the number of parallel processes increasing. A good indicator is the number of generated registers. Indeed, with n registers, we can implement 2^n states in an automaton.
The chosen process is very simpler, not to disturb the result:

```
module WIO:
Input: I; Output: O;
wait I >> emit O
end
```

which waits the I signal and emits the signal O one time as soon as I occurs. Here is the obtained table by the figure 5:

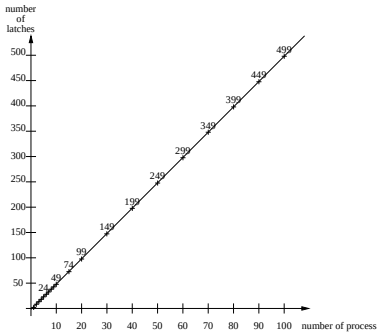


Figure 5: Evolution of the Registers number

The relation between numbers of processes and number of registers seems to be linear, that is an excellent thing! The linear observed factor of 5 is only characterized by the equational semantic of `parallel` and `run` statements. In a next equationnal semantic, this number should be reduced.

6 Example

We illustrate LE usage on an industrial example concerning the design of a mecatronics process control: a pneumatic prehensor. We first describe how the system works. Then we present the system implementation with LE language. Finally, simulation and verification are performed.

6.1 Mecatronics System Description

A pneumatic prehensor takes and assembles cogs and axes. The physical system mainly consists of two double acting pneumatic cylinders and a suction pad. This example has been taken as a

benchmark by an automation specialist group[7], to experiment new methods of design and analysis of *discrete event systems*. The (U cycle) kinematics of the system is described in Fig.6. Note that the horizontal motion must always be done in the high position.

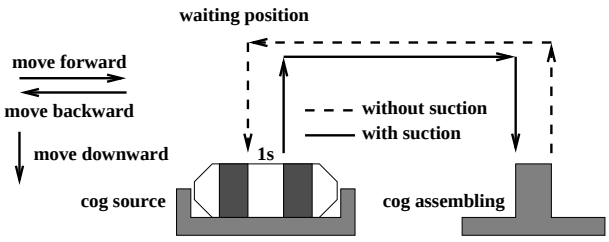


Figure 6: A pneumatic prehensor

The horizontal motion pneumatic cylinder is driven by a bistable directional control valve (bistable DCV). The associated commands are `MoveFor` (short for move forward) and `MoveBack` (short for move backward). The vertical cylinder is driven by a monostable directional control valve 5/2 whose active action is `MoveDown` (move downward). In the absence of activation, the cylinder comes back to its origin position (high position). The suction pad (`SuckUp` command) is activated by a monostable DCV (the suction is done by a Venturi effect).

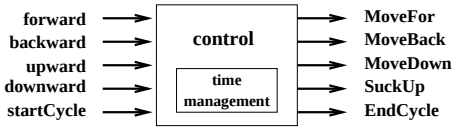


Figure 7: Input/output signals

6.2 Mecatronics System LE Implementation

In what follow we consider the control part of the system. Fig.7 gathers incoming information (from the limit switches associated with the cylinders) and outgoing commands (to the pre-actuators). To implement this application in LE language, we adopt a top down specification technique. At the highest hierarchical level , the controller is the parallel composition of an initialization part followed by the normal cycle running and a temporisation module. This last is raised by a signal `start_tempo` and emits a signal `end_tempo` when the temporisation is over. Of course, these two signals are not in overall interface of the controller, they are only use to establish the communication between the two parallel sides. The following LE program implements the high level part of the controller:

```
module Control:
```

[7] `http://www.lurpa.ens-cachan.fr/cosed`

```
Input:forward, backward, upward,  downward,
      StartCycle;
Output:MoveFor, MoveBack, MoveDown, SuckUp,
       EndCycle ;

Run: "./TEST/control/" : Temporisation;
     "./TEST/control/" : NormalCycle;

local start_tempo, end_tempo {
   { wait upward >> emit MoveFor
     >> wait backward >> run NormalCycle
   }
 ||
   { run Temporisation}
}
end
```

The second level of the specification describes temporisation and normal cycle phasis. Both Temporisation and NormalCycle modules are defined in external files. Temporisation module performs a delaying operation (waiting for five successive reactions and then emitting a signal `end_tempo`. The overall LE code is detailed in appendix C. In this section, we only discuss the NormalCycle module implementation. NormalCycle implementation is a loop whose body specifies a single cycle. According to the specification, a single cycle is composed of commands to move the pneumatic cylinders with respect to their positions and a call to a third level of implementation (Transport) to specify the suction pad activity.

```
module Transport :

Input: end_tempo, upward, forward, downward;
Output: MoveFor, MoveDown, SuckUp;

local exitTransport {

    { emit MoveDown >> wait end_tempo
      >> wait upward >> emit MoveFor
      >> wait forward >> emit MoveDown
      >> wait downward >> emit exitTransport
    }
 ||

    abort
      { loop { pause >> emit SuckUp }}
    when exitTransport
}
```

```
end

module NormalCycle :

Input:  StartCycle, downward, upward, backward,
        forward, end_tempo;
Output: start_tempo, MoveDown, MoveBack,
        MoveFor, SuckUp, EndCycle;

{ present StartCycle { nothing} else wait StartCycle}

>>

{
  loop { emit MoveDown
         >> wait downward >> emit start_tempo
         >> run Transport
         >> wait upward >> emit MoveBack
         >> wait backward >> emit EndCycle }
}
end
```

To compile the overall programs, we performed a separate compilation: first, Temporisation and NormalCycle modules have been compiled and respectively saved in `lec` format file. Second, the main Control module has been compiled according to our compilation scheme (see figure 4).

6.3 Mecatronics System Simulation and Verification

To check the behavior of our implementation with respect to the specification, we first simulate it and then perform model-checking verification. Both simulation and verification relies on the generation of `blif` format from `clem` compiler.
Figure 8 shows the result of Control simulation with a graphical tool we have to simulate blif format modules.
On another hand, to formally prove safety properties we rely on model checking techniques. In this approach, the correctness of a system with respect to a desired behavior is verified by checking whether a structure that models the system satisfies a formula describing that behavior. Such a formula is usually written by using a temporal logic. Most existing verification techniques are based on a representation of the concurrent system by means of a labeled transition system (LTS). Synchronous languages are well known to have a clear semantic that allows to express the set of behaviors of program as LTSs and thus model checking techniques are available. Then, they rely on formal methods to build dependable software. The same occurs for LE language, the LTS model of a program is naturally encoded in its equational semantic.
A verification means successfully used for synchronous formalisms is that of observer monitoring [10]. According to this technique, a safety property ϕ can be mapped to a program Ω which runs in parallel with a program P and observes its behavior, in the sense that at each instant Ω reads

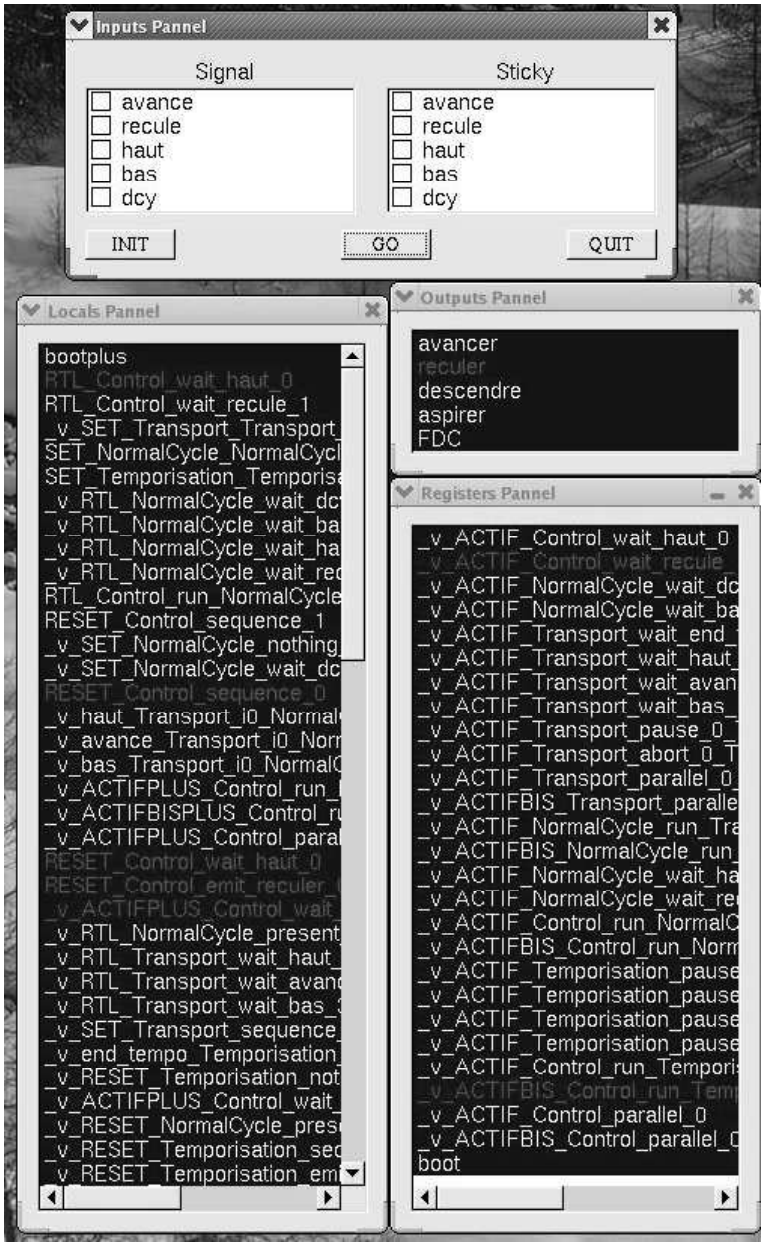


Figure 8: Control module simulation panels

both inputs and outputs of P. If Ω detects that P has violated ϕ then it broadcasts an "alarm" signal. As a consequence, we can rely on model checking based tools to verify property of LE language. But, our approach provides us with separate compilation and requires to be completed by a modular verification. We aim at proving safety properties are preserved through LE language operator application.
To verify that the suction is maintained from the instant where the cycle begins up to the cycle ends, the following observer can be written in LE .

```
module CheckSuckUp;
Input SuckUp, S;
Output exitERROR;
present SuckUp
        { present S {nothing} else {wait S}}
        else {pause>>emit exitERROR}
end

module SuctionObs:

Input:forward, backward, upward,  downward,
      StartCycle, Output:MoveFor, MoveBack,
      MoveDown, SuckUp, EndCycle ;
Output: ERROR;

local exitERROR {
 abort {
  loop {
   present StartCycle {nothing}
           else {wait StartCycle} >>
   present MoveDown {nothing}
                    else {wait MoveDown}  >>
   present downward {nothing}
                    else { wait downward} >>
   present MoveDown {nothing}
           else { present SuckUp
                     {run CheckSuckUp[upward\S]   >>
                      run CheckSuckUp[MoveFor\S]  >>
                      run CheckSuckUp[forward\S]  >>
                      run CheckSuckUp[MoveDown\S] >>
                      wait downward
                     }
                     else {emit exitERROR}
                }
  }
 when exitERROR >> emit ERROR
 }
```

```
}
end
```

To specify the observer we first define a module (CheckSuckUp) which checks wether the signal `SuckUp` is present and goes in the state where signal `S` is present. If `SuckUp` is absent , exitERROR is emitted. Calling this module, the observer tests the presence of signal SuckUp in each possible states reached when cylinders move.
To achieve the property checking, we compile a global module made of the Control module in parallel with the SuctionObs module and we rely on model checker to ensure that ERROR is never emitted. By the time, we generate the BLIF format back end for the global module and we call `xeve` model-checker [4] to perform the verification. In the future, we intend to interface NuSMV [5] model-checker.
The chosen example is a very simple one but we hope understandable in the framework of a paper. Nevertheless, we compiled it globally and in a separate way. The global compilation takes about 2.7 s while the separate one takes 0.6 s on the same machine. We think that it is a small but promising result.

7 Conclusion

In this work, we have presented a new synchronous language LE that supports separate compilation. We defined its behavioral semantic giving a meaning to each program and allowing us to rely on formal methods to achieve verification. Then, we also defined an equational semantic to get a means to really compile programs in a separate way. Actually, we have implemented the `clem/clef` compiler. This compiler is the core of the design chain (see section 5.3.3) we have to specify control-dominated process from different front-ends: a graphical editor devoted to automata drawing, or direct LE language specification to several families of back-ends:

- *code generation*: we generate either executable code as C code or model-driven code: Esterel, Lustre code for software applications and Vhdl for harware targets.
- *simulation tools*: thanks to the *blif* format generation we can rely our own simulator (*blif_simul*) to simulate LE programs.
- *verification tools*: BLIF is a well-suited format to several model-checkers(xeve, sis) and has its automata equivalence verifier (blif2autom, blifequiv).

In the future, we will focus on three main directions. The first one concerns our compilation methodology. Relying on an equational semantic to get modular compilation could lead to generate inefficient code. To avoid this drawback, we plan to study others equational semantic rules (in particular for parallel and run statements) more suited for optimization. The second improvement we aim at, is the extension of the language. To be able to deal with control-dominated systems with data (like sensor handling), we will extend the syntax of the language on the first hand. On the other hand, we plan to integrate abstract interpretation techniques (like polyhedra intersection, among others) [6] to take

into account data constraints in control. Moreover, we also need to communicate with signal processing or automation world through their specific tool Matlab/Simulink (http://www.mathworks.com). Another language extension is to allow a bound number of parallel operators. This extension is frequently required by users to specify their applications. Semantic rules for this new bound parallel operator cannot be straightly deduced from the actual rules we have, and require a deep change but then would improve LE expressiveness. Finally, we are interested in improving our verification means. The synchronous approach provides us with well-suited models to apply model checking techniques to LE programs. The more efficient way seems to directly interface a powerful model-ckecker (as NuSMV [5]) and to be able to run its property violation scenarios in our simulation tool. Moreover, our modular approach opens new ways to modular verification. We need to prove that LE operators preserve properties: if a program P verify a property ϕ, then all program using P should verify a property ϕ' such that the "restriction" of ϕ' to P implies ϕ.

References

[1] C. André, H. Boufaïed, and S. Dissoubray. Syncharts: un modèle graphique synchrone pour système réactifs complexes. In *Real-Time Systems(RTS'98)*, pages 175–196, Paris, France, January 1998. Teknea.

[2] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, available at: http://www.esterel-technologies.com 1996.

[3] G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stearling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honor of Robin Milner*. MIT Press, 2000.

[4] Amar Bouali. Xeve , an esterel verification environment. Technical report, CMA-Ecole des Mines, 1996.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: an OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceeeding CAV*, number 2404 in LNCS, pages 359–364, Copenhagen, Danmark, July 2002. Springer-Verlag.

[6] P. Cousot and R. Cousot. On Abstraction in Software Verification. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceeeding CAV*, number 2404 in LNCS, pages 37,56, Copenhagen, Danmark, July 2002. Springer-Verlag.

[7] S.A. Edwards. Compiling esterel into sequential code. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES 99)*, pages 147–151, Rome, Italy, May 1999.

[8] M. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[9] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[10] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[11] D. Harel and A. Pnueli. On the development of reactive systems. In *NATO, Advanced Study institute on Logics and Models for Verification and Specification of Concurrent Systems*. Springer Verlag, 1985.

[12] C. Huizing and R. Gerth. Semantics of reactive systems in abstract time. In *Real Time: Theory in Practice, Proc of REX workshop*, pages 291–314. W.P. de Roever and G. Rozenberg Eds,LNCS, June 1991.

[13] D. Potop-Butucaru and R. De Simone. *Formal Methods and Models for System Design*, chapter Optimizations for Faster Execution of Esterel Programs. Gupta, P. LeGuernic, S. Shukla, and J.-P. Talpin, Eds ,Kluwer, 2004.

[14] T. Shiple. *Formal Analysis of Cyclic Circuits*. PhD thesis, University of California, 1996.

[15] Esterel Technologies. Esterel studio suite, www.estereltechnologies.com.

[16] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of esterel for real-time embedded systems. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems,*, pages 2–8, San Jose, California, United States, November 2000.

A LE Grammar

In this appendix, we describe the complete grammar of the LE language. That description supports the following agreements:

- <> notation represents tokens, for instance <module> represents the name *module*;
- two specific tokens are introduced: IDENT for identifier and STRING to denote a usual string;
- the notation $\star$ and $+$ are used for repetition: $signal\_name\star$ means a number of $signal\_name$, possibly 0,while $signal\_name+$ means at least one occurrence;
- the single charater are straighly written (as {,},[,], and \).
- the character # denotes the empty word;

```
program: <module> module_name ':' module_interface module_body <end>;

module_interface : input_signal_list output_signal_list run_decl_list;

input_signal_list : # | <Input:> signal_name+ ';' ;
output_signal_list : #  | <Output:> signal_name+ ';' ;
run_decl_list : #  | <Run:> run_declaration+ ;
run_declaration: path ':' module_name;

module_body : instruction | automaton ;

instruction : statement | '{' instruction '}' ;

statement : parallel
          | sequence
          | present
          | loop
          | wait
          | emit
          | abort
          | nothing
          | pause
          | halt
          | local
          | run
          ;

parallel : instruction '||' instruction ;
sequence :  instruction '>>' instruction ;
present : <present> xi_expression  instruction   <else> instruction ;
```

```
loop :  <loop> '{' instruction '}' ;
wait :   <waitL>  signal_name ;
emit : <emit> signal_name ;
abort : <abort> '{' instruction '}' <when> signal_name ;
pause : <pause> ;
nothing : <nothing> ;
halt : <halt> ;
local :  <local> signal_name+ '{' instruction '}'
run : <run> module_name  renaming ;
renaming : #  | '[' single_renaming+ ']' ;
single_renaming :  signal_name '\' signal_name


automaton : <automaton> state+ transition_def ;
state : <state> state_name opt_final opt_run  action ';' ;
opt_run: #  | run ;
transition_def :  <transition> transition+ ;
transition : opt_initial opt_final  opt_source_state  trigger  action opt_target_state ;
opt_source_state: # | state_name;
opt_target_state: # | '->' state_name;
opt_initial : # | <initial> ;
opt_final:  # | <initial> ;
trigger:  # | xi_expression ;
action: # | '/' signal_name+ ;
xi_expression : xi_expression <or> xi_expression
              | xi_expression <and> xi_expression
              | <not> xi_expression
              | '{' xi_expression '}'
              | signal_name
              ;

signal_name : IDENT;
module_name : IDENT;
path : STRING;
```

B PERT Algorithms

B.1 First Step of PERT ALGORITHM

The following algorithm is the first step of the overall PERT algorithm we implement. It builds a forest of variable dependency trees.

```
for each equation xi=fi(...,xj,...)
begin
   for all j needed by fi
   begin
      Upstream[i].add(j);
      Downstream[j].add(i)
   end
end
```

B.2 Second Step of PERT Algorithm

The second step of the PERT algorithm we implement consists in the propagation of the *Early* Dates from the inputs and the registers, to the outputs. Similarly, the *Late* Dates are propagated from the outputs to the inputs and the registers according to the following algorithm:

```
for each variable id i
begin
   if(Upstream[i] = empty set)
   begin
      /* final output */
      late[i]=0
      for each j in Downstream[i]
      begin
         late_propagation(j,1)
      end
   end
   if(Downstream[i] = empty set)
   begin
      /* real input or constante */
      early[i]=0
      for each j in Upstream[i]
      begin
         early_propagation(j,1)
      end
   end
end

function late_propagation(id,date)
begin
```

```
  if(late[id] < date)
  begin
    late[id]=date
    for each j in Dowstream[id]
    begin
       late_propagation(j,date+1)
    end
  end
end

function early_propagation(id,date)
begin
  if(early[id] < date)
  begin
    early[id]=date
    for each j in Upstream[id]
    begin
       early_propagation(j,date+1)
    end
  end
end
```

C LE Control Example Code

In this appendix, we detail the LE code for the *Control* example described in section 6.

C.1 Control Module Specification

The main file of the *Control* example is *Control.le*. We give its content:

```
;;=================================================
;; LE specification for a mecatronic system
;; Main file: Control specification
;;=================================================


module Control:

Input:  forward, backward, upward, downward, StartCycle;
Output: MoveFor, MoveBack, MoveDown, SuckUp, EndCycle;

Run: "/home/ar/GnuStrl/work-ar/TEST/control/" : Temporisation;
     "/home/ar/GnuStrl/work-ar/TEST/control/" : NormalCycle;

local start_tempo, end_tempo {

     { wait upward >> emit MoveBack >> wait backward >> run NormalCycle}
  ||
     { run Temporisation}
}

end
```

The *Control* module calls two external modules *Temporisation* and *NormalCycle*. The paths to *Temporisation.le* and *NormalCycle.le* files where the respective LE codes of these called modules are, is given in *Control* module interface. During compilation, a file *temporisation.lec* (resp *NormalCyle.lec*) is searched in the compilation library. If *Temporisation* (resp *NormalCycle*) has not been already compiled then it is compiled. Thus, in both cases, the compiled code is included in *Control* module code.

C.2 Temporisation module Specification

```
;;=================================================
;; LE specification for a mecatronic system
;; Temporisation  specification
;;=================================================
```

```
module Temporisation :

Input:  start_tempo;
Output: end_tempo;

present start_tempo {
 pause >> pause >> pause >> pause >> emit end_tempo }
else nothing

end
```

C.3 NormalCycle module Specification

```
;;=====================================================
;; LE specification for a mecatronic system
;; Normal cycle  specification
;;=====================================================

module Transport :

Input: end_tempo, upward, forward, downward;
Output: MoveDown,MoveFor, SuckUp;

local exitTransport {

    { emit MoveDown >> wait end_tempo >> wait upward >> emit MoveFor
      >> wait forward >> emit MoveDown >> wait downward  >> emit exitTransport
    }
 ||

    abort
      { loop { pause >> emit SuckUp }}
    when exitTransport
}

end

module NormalCycle :

Input: StartCycle, downward, upward, backward, end_tempo, forward;
Output: start_tempo, MoveDown,MoveBack, EndCycle, MoveFor, SuckUp;

{ present StartCycle { nothing} else wait StartCycle }
```

```
>>

{
  loop { emit MoveDown >> wait downward >> emit start_tempo  >> run Transport
         >> wait upward >> emit MoveBack
         >> wait backward >> emit EndCycle }
}

end
```

The *NormalCycle* module called itself a *Transport* module, but contrary to *Control* module, the specification of the called module is given in the same file. Thus, no path has to be supplied in *NormalCycle* interface.

D Condition Law Expansion

In this appendix, we discuss how a term from ξ algebra resulting of the application of the condition law is expanded in a pair of boolean values in $\mathbb{B}$.. Let us consider a ξ term X. We recall that X is isomorphic to a pair of boolean (X_{def}, $X_{va;}$) (see section 3.1) and we want to prove the following equalities: $(X \blacktriangleleft c)_{def} = X_{def}.c$ and $(X \blacktriangleleft c)_{val} = X_{val}.c$, where $c \in \mathbb{B}$.
These equalities are very useful for implementing the condition law in the compilation phasis.
First, relying on the definition of the isomorphism between ξ algebra and $\mathbb{B} \times \mathbb{B}$, we can expand the encoding of the condition law as follow:

X	X_{def}	X_{val}	c	$(X \blacktriangleleft c)$	$(X \blacktriangleleft c)_{def}$	$(X \blacktriangleleft c)_{val}$
1	1	1	0	$\bot$	0	0
0	1	0	0	$\bot$	0	0
$\top$	0	1	0	$\bot$	0	0
$\bot$	0	0	0	$\bot$	0	0
1	1	1	1	1	1	1
0	1	0	1	0	1	0
$\top$	0	1	1	$\top$	0	1
$\bot$	0	0	1	$\bot$	0	0

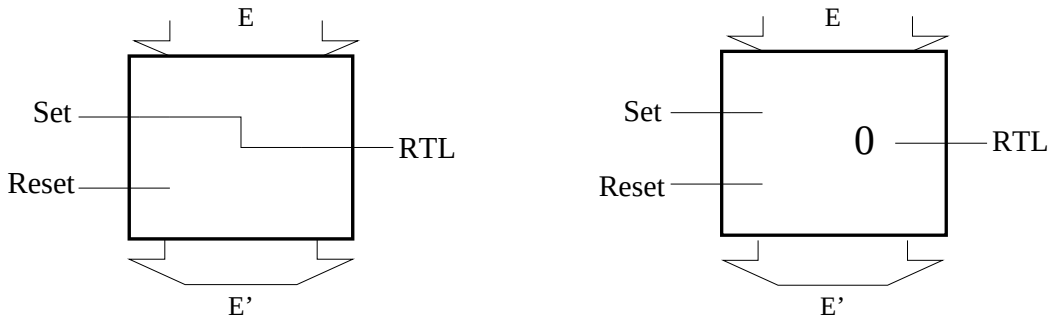
where $c \in \mathbb{B}$.
Thus, we can deduce:

$$\begin{aligned}(X \blacktriangleleft c)_{def} &= X_{def}.X_{val}.c \ + \ X_{def}.\overline{X_{val}}.c \\ &= X_{def}.c.(X_{val} + \overline{X_{val}}) \\ &= X_{def}.c\end{aligned}$$

$$\begin{aligned}(X \blacktriangleleft c)_{val} &= X_{def}.X_{val}.c \ + \ \overline{X_{def}}.X_{val}.c \\ &= X_{val}.c.(X_{def} + \overline{X_{def}}) \\ &= X_{val}.c\end{aligned}$$

E LE Statement Circuit Description

In this appendix, we show the circuits corresponding to LE statement. We rely on them to compute the equational semantic of each LE operator.



(a) Circuit for `nothing`

(b) Circuit for `halt`

Figure 9: Basic LE statements circuit semantic

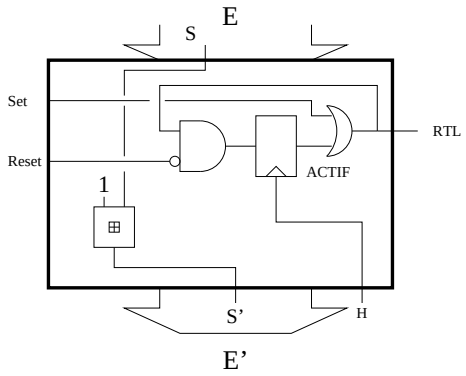
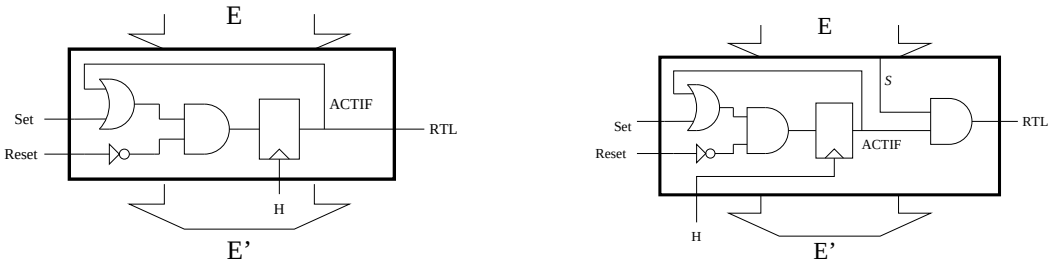


Figure 10: Circuit for `emit`S



(a) Circuit for `pause` (b) Circuit for `wait`

Figure 11: Pause and Wait LE statements circuit semantic

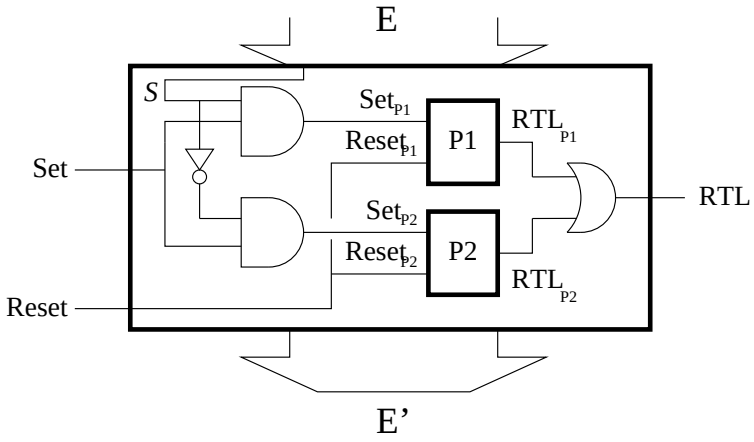


Figure 12: Circuit for `Present` $S\{\mathrm{P}_1\}$`else`$\{\mathrm{P}_2\}$

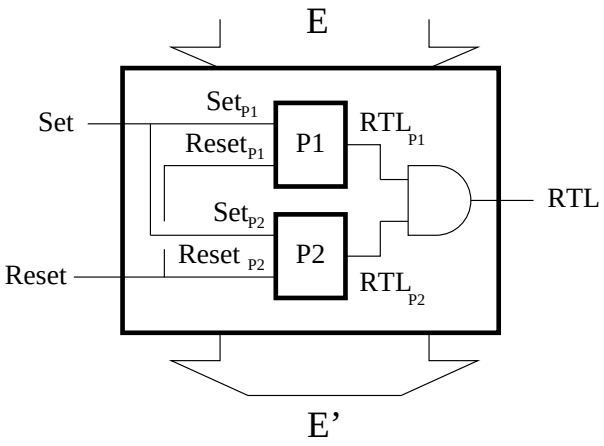


Figure 13: Circuit for$\mathrm{P}_1 \| \mathrm{P}_2$

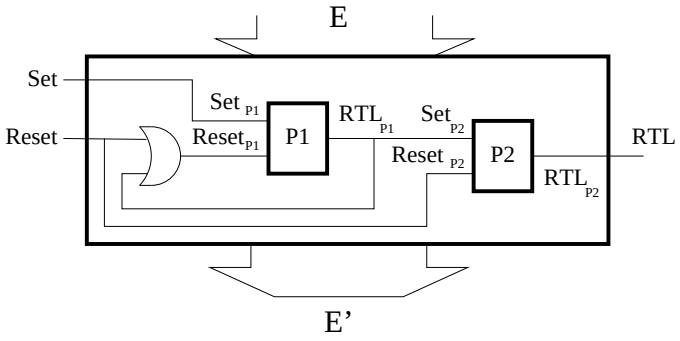


Figure 14: Circuit for $\mathrm{P}_1 \gg \mathrm{P}_2$

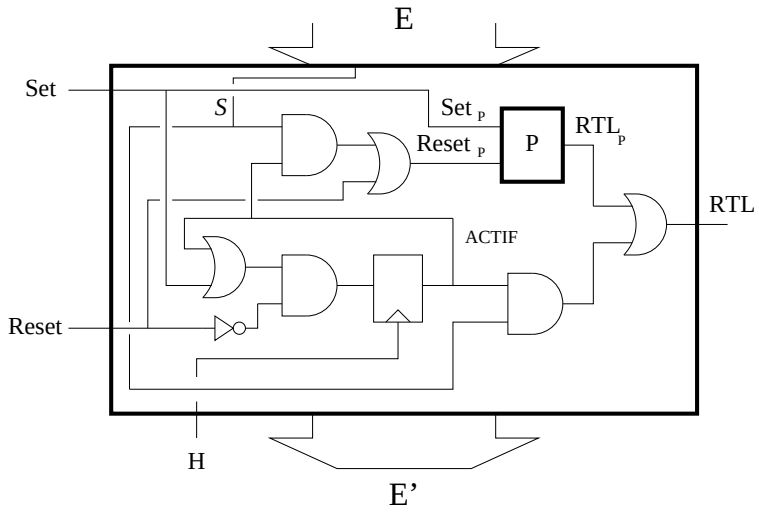


Figure 15: Circuit for abort P when S

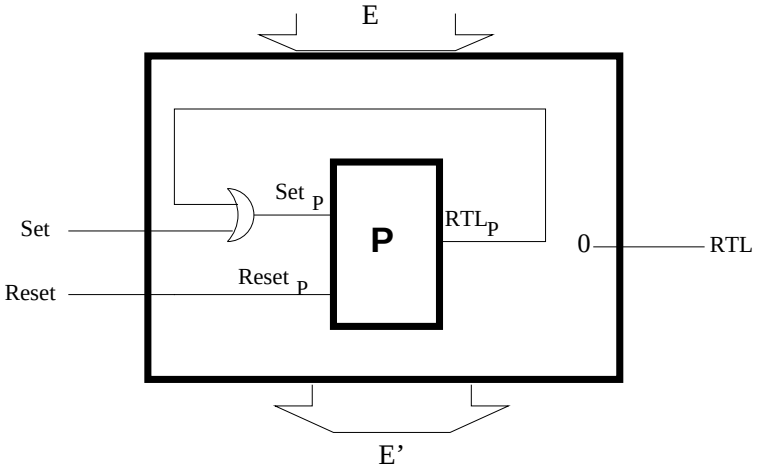


Figure 16: Circuit for loop {P}

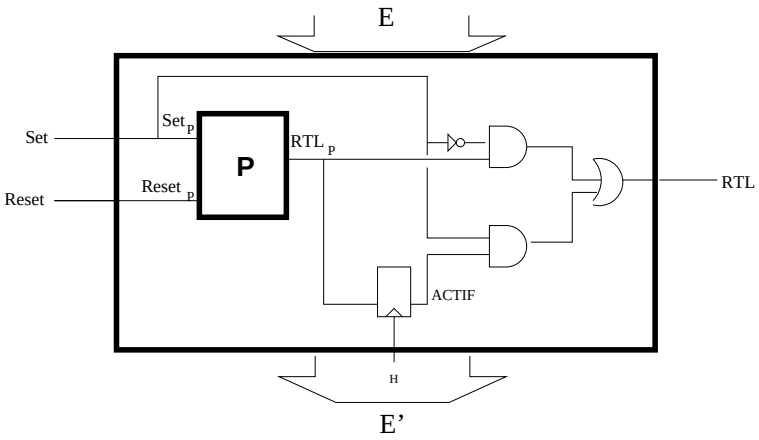


Figure 17: Circuit for `run{P}`

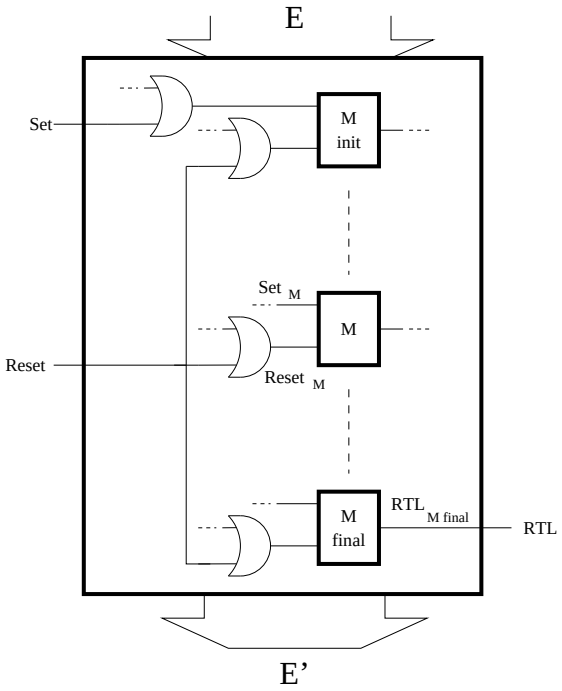


Figure 18: Circuit for $\mathcal{A}(\mathrm{M_{init}}, .., \mathrm{M}, ..., \mathrm{M_{final}})$

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399