HAL
archives-ouvertes.fr

Synchronous Automata For Activity Recognition

Ines Sarray, Annie Ressouche, Sabine Moisan, Jean-Paul Rigault, Daniel Gaffé

▶ **To cite this version:**

Ines Sarray, Annie Ressouche, Sabine Moisan, Jean-Paul Rigault, Daniel Gaffé. Synchronous Automata For Activity Recognition. [Research Report] RR-9059, Inria Sophia Antipolis. 2017. hal-01505754

HAL Id: hal-01505754
https://hal.inria.fr/hal-01505754

Submitted on 19 Apr 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Synchronous Automata For Activity Recognition

Ines SARRAY, Annie Ressouche, Sabine Moisan, Jean-Paul Rigault, Daniel Gaffé

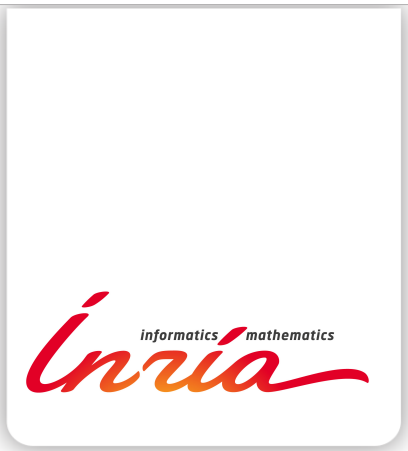
RESEARCH REPORT

N° 9059

April 2017

Project-Team Stars

ISSN 0249-6399 ISRN INRIA/RR--9059--FR+ENG



Synchronous Automata For Activity Recognition

Ines SARRAY*, Annie Ressouche†, Sabine Moisan‡, Jean-Paul Rigault§, Daniel Gaffé¶

Project-Team Stars

Research Report n° 9059 — April 2017 — 28 pages

Abstract: Activity recognition is important for security and safety in many domains, such as surveillance and health care. We propose to describe activities as a series of actions, triggered and driven by environmental events. We rely on *synchronous automata* to describe such activities. We chose the synchronous paradigm because it has a well-founded semantics and it ensures determinism and parallel composition. Moreover, we already developed tools that can be adapted to activity recognition. In this report we propose a new synchronous language to express synchronous automata, that relies on a formal semantics and that allows us to perform model-checking proofs, to compile activities into equation systems, and to automatically generate the corresponding recognition code.

Key-words: activity recognition, synchronous automata,synchronous language, algebra, formal semantics

* ines.sarray@inria.fr
† annie.ressouche@inria.fr
‡ sabine.moisan@inria.fr
§ jean-paul.rigault@inria.fr
¶ daniel.gaffe@unice.fr

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Les automates synchrones pour la reconnaissance d'activités

Résumé : La reconnaissance d'activité est devenue de plus en plus importante dans différents domaines comme la surveillance et la santé, pour sa sécurité et sûreté. Nous proposons de décrire les activités en tant que série d'actions déclenchées et pilotées par des événements provenant de l'environnement, et nous utilisons les automates synchrones pour les représenter. Nous avons choisi le paradigme synchrone pour sa sémantique bien fondée et parce qu'il assure le déterminisme et la composition parallèle. De plus, nous avons déjà développé des outils synchrones qui peuvent être adaptés à la reconnaissance d'activité. Nous proposons aussi un nouveau langage synchrone pour exprimer ces automates synchrones. Ce langage est basé sur une sémantique formelle qui permet de vérifier et valider nos modèles d'activités à reconnaitre à l'aide des preuves du model-checking, de les compiler dans des systèmes d'équations, et de générer automatiquement son code de reconnaissance correspondant.

Mots-clés : reconnaissance d'activités, automates synchrones, langage synchrone, algèbre, sémantiques formelles

Contents

1 Introduction

For security and safety reasons, many applications require to recognize various activities, corresponding to interesting behaviors in the application domain. Activity Recognition aims at recognizing a sequence of actions that follow a predefined model of an activity[1]. We mainly work on video-surveillance and monitoring applications; in such applications, the data coming from sensors (video-cameras, etc.) are first processed to recognize and track objects and to detect low-level events. This low-level information is collected and transformed into high-level data (a.k.a. events). We are mainly interested in medical applications to help physicians to detect abnormal behaviors. We propose to describe activities by finite automata, since automata are simple and "natural" means to describe behaviors. There are several models of automata, and because we have worked with reactive systems for a long time, we chose *synchronous* finite automata . The Synchronous Paradigm relies on a discrete logical time composed of a sequence of logical instants, defined by the system reactions. Synchronous systems have many interesting properties such as determinism, parallelism, and formal verification. We also chose this kind of representation because we have experience in this field and we have already developed tools to formalize, prove and recognize activities.

This report is organized as follows: the next section introduces the synchronous model of automata and describes our activity description language. Then, in section 3, we introduce the semantics and the mathematical concepts on which we rely to define and verify the behavior of programs and to compile them, before concluding in section 4.

[1] Several authors use "scenario recognition" instead of "activity recognition". Strictly speaking, an activity may include choices, loops, concurrency whereas a scenario is just a trace (an instance) of the activity. Nevertheless, in this report we use equally both terminologies.

2 Synchronous Automata

There are several models of automata and, for each model, several possible representations. We choose the synchronous model and we propose a textual user dedicated language to define the automata.

2.1 Synchronous Model of Automata

Reactive systems listen to input events coming from the external environment and react to them by generating output events towards the environment. Such systems can be complex. The *synchronous model* is a way to reduce the complexity of behavior description by considering their evolution along successive discrete *instants*. An instant starts when some input events are available. All these events are frozen; the output and internal events deriving from these inputs are computed until stability (fixed point) is achieved; the instant finishes by delivering the output events to the environment. No inputs occurring "during" the instant are considered. Hence, instants are atomic, their sequence defines a logical time[2].

The synchronous paradigm is interesting because it ensures determinism and it supports concurrency through deterministic parallel composition. It is also well-founded, it relies on a well-defined semantics, and many tool sets have been developed for simulation, verification, and code generation of synchronous automata.

Synchronous models can be represented as *Mealy machines*. The Mealy machines that we consider are 5-uples of the form $< Q, q_{init}, I, O, T >$, where Q is a finite set of states, $q_{init} \in Q$ is the initial state, I (resp. O) is a finite set of input (resp. output) events; $T \subseteq (Q \times I) \times (Q \times O)$ is the transition relation.

This is an explicit representation of Mealy machines as automata. Mealy himself introduced an other representation as Boolean equation systems that calculate both the output event values and the next state from the input events and the current state [6]. We call this representation "implicit" Mealy machines.

For the users to describe synchronous automata, synchronous languages such as Lustre, Esterel, Scade and Signal [5] have been defined. These languages are for expert users. We propose another synchronous language that is easier to understand and use for non computer scientists (e.g., physicians). We call it ADeL (Activity Description Language) and we describe it in the next section.

2.2 Activity Description Language

ADeL is a modular and hierarchical language, which means that an activity may contain one or more sub-activities. The description of an activity consists of several parts: first the user defines the types of the activity participants, their roles, and the initial state.

In the body, the user describes the expected behavior using a set of control operators. These operators are the base of the ADeL language, they deal with events coming from the external environment. Some of these operators are instantaneous and others take at least one instant to process. By composing these operators (detailed in Table 1) one can build complex instructions, describing sub-scenarios.

2.3 Use Case

We present an example of the ADeL language. This program describes the activity of a person (e.g., suffering from Alzheimer) answering a phone call. Note that we consider the regular "clock time" (e.g., min) as just another input event so that ADeL operators (such as **timeout**) can cope with it. Figure 1 shows the corresponding automaton.

[2]In this model, instants take "no time" with respect to the logical time they define.

nothing	does nothing and terminates instantaneously.
[**wait**] S	waits for event S and suspends the execution of the scenario until S is present; operator wait can be implicit or explicit.
p_1 **then** p_2	starts when p_1 starts; p_2 starts when p_1 ends; the sequence terminates when p_2 does.
p_1 **parallel** p_2	starts when p_1 or p_2 starts; ends when both have terminated.
p_1 **during** p_2	p_1 starts only after p_2 starts and must finish before p_2 ends.
while *condition* $\{p\}$	p is executed only if the *condition* is verified. When p ends, the loop restarts until the *condition* holds.
stop $\{p\}$ **when** S **alert** S_1	executes p to termination as long as S is absent, otherwise when S is present, aborts p, sends an alert S_1, and terminates.
if *condition* **then** p_1 **else** p_2	executes p_1 if *condition* holds, otherwise executes p_2.
p **timeout** S $\{p_1\}$ [**alert** S_1]	executes p; stops if S occurs before p terminates and possibly sends alert S_1; otherwise executes p_1 when p has terminated.
alert S	raises an alert.
local(*events*) $\{p\}$	declares internal events to communicate between sub parts of p.
call(*scenario*)	calls a sub-scenario.

Table 1: ADeL operators. S, S_1 are events (received or emitted); p, p_1 and p_2 are instructions; *condition* is either an event or a Boolean combination of event presence/absence.

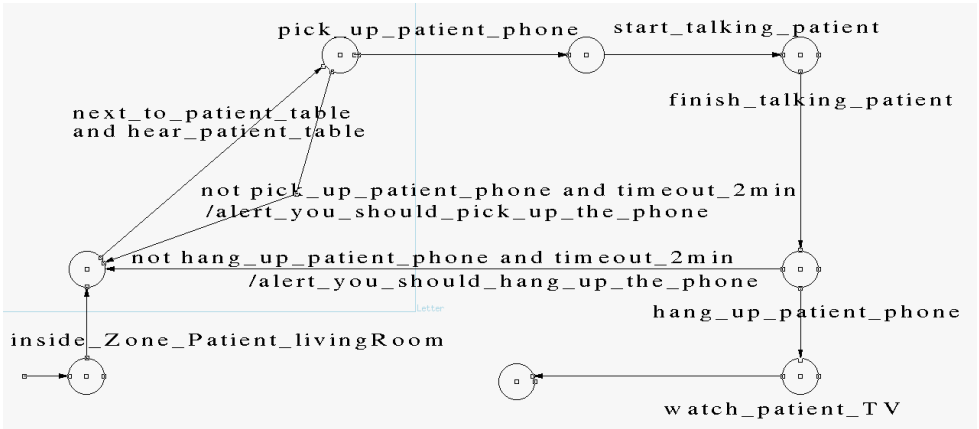


Figure 1: Synchronous automaton of the *phoneCall* program. In this explicit representation, transitions bear labels interpreted as follows: the left part of the label (before the "/" sign) is the trigger and the right part represents the output events.

```
Type Person, Equipment, Zone;
Scenario phoneCall:
Roles
     patient: Person;
     phone: Equipment;
     table: Equipment;
     TV: Equipment;
     livingRoom: Zone;
Subscenarios
    next_to(Person, Equipment);
    hear(Person, Equipment);
    pick_up(Person, Equipment);
    start_talking(Person);
    finish_talking(Person);
    hang_up(Person, Equipment);
    watch(Person, Equipment);
InitialState : inside_Zone(patient, livingRoom);
Start
    next_to(patient, table) || hear(patient, phone)
    then
    pick_up(patient, phone) timeout 2.0min
      {
            start_talking(patient)
            then
            finish_talking(patient)
            then
            hang_up(patient, phone) timeout 2.0min
                {
                     watch(patient, TV)
                }
            alert you_should_hang_up_the_phone
       }
   alert you_should_pick_up_the_phone
End
```

3 Semantics and Compilation

To provide the language with sound foundations we turn to a formal semantic approach. First, conditional rewriting rules are a classical and rather natural way to formally express the intuitive semantics. This form of *behavioral semantics* gives an abstract description of a program behavior and facilitates its analysis. However it is not convenient as an implementation basis nor suitable for proofs (e.g., model-checking). Hence we also define an *equational semantics* which maps an ADeL program into a Boolean equation system representing its automaton. The ADeL compiler can easily translates this equation system into an efficient code. Using such a double semantics is somewhat traditional in the synchronous language area [2].

Since we have two different semantics, it is mandatory to establish their relationship. In fact we want to prove that what is executed–on the basis of the equational semantics–also conforms to an equivalent behavioral semantics.

3.1 Mathematical Context

The ADeL semantics rely on the notion of an environment which is a finite set of events. An event contains the information about its status (that is its presence). Environments memorize the status of their events in each instant. To formally define the ADeL semantics, we represent the event status with an algebra structure.

3.1.1 The ξ 4-valued Algebra

We use a 4-valued algebra ($\xi = \{\bot, 0, 1, \top\}$) to represent the status of events. $\bot$ means that the status of the event is not determined, 0 that the event is absent, 1 that the event is present, and $\top$ that the status of the event can not be determined because it has two incompatible status in the same instant (e.g., it has the values 0 and 1 in different parts of the program).

The semantics we will propose to rely on for both verification and compilation purposes, associates a ξ equation system to each program instead of a Boolean one as other semantics for synchronous language do. In each reaction, the equation system helps us to compute an output environment from an input one. This means that at least we must supply ξ with the usual logical operators: $\neg$, $\boxplus$, $\boxdot$ which should have a Boolean like definition to be able to express the status of output and local signals from input and local signal status. Moreover, for the parallel operator ($P_1 \| P_2$), the overall equation system must be deduced from the respective equation systems of P_1 and P_2. In particular, when the same event has a status in each equation systems, in the resulting equation system, its status should be the "unification" of the information of its respective status in P_1 and P_2 equation systems. For instance, assume that event S has the status $\bot$ in P_1 equation system and the status 1 in P_2 equation system, then it should have 1 has status in $P_1 \| P_2$ equation system. Thus, we must also supply ξ with such an operation, called *Unify* ($\sqcup$) which perform the union of information concerning an event in different environments. On another hand, the semantics computes the unique least fixed point $E = F(E)$ in the 4-valued algebra considered. To ensure that least fixed points exist and can be computed, we need a 4-valued algebra with operators making F monotonic. To this aim, we will also consider the symmetric operator of $\sqcup$, called $\sqcap$.

Thus to fill these requirements, we supply ξ with a bilattice structure [4]. Bilattices are mathematical structures having two distinct partial orders denoted $\leq_B$ and $\leq_K$ and a $\neg$ operation. $\leq_B$ represents an extension of the usual Boolean order and $\leq_K$ expresses the level of information about the presence of an event:

Definition 1. (Ginsberg[4]) A **bilattice** is a structure ($\mathscr{B}$, $\leq_B$ $\leq_K$, $\neg$) consisting of a non empty set $\mathscr{B}$, partial orderings $\leq_B$ and $\leq_K$ and a mapping $\neg : \mathscr{B} \mapsto \mathscr{B}$ such that:

1. ($\mathscr{B}$, $\leq_B$) and ($\mathscr{B}$, $\leq_K$) are complete lattices
2. $x \leq_B y \Rightarrow \neg y \leq_B \neg x$, $\forall x, y \in \mathscr{B}$
3. $x \leq_K y \Rightarrow \neg x \leq_K \neg y$, $\forall x, y \in \mathscr{B}$
4. $\neg\neg x = x$, $\forall x \in \mathscr{B}$

In ξ, we define two orders as follows:

$\bot$	$\leq_K$	0	$\leq_K$	$\top$
$\bot$	$\leq_K$	1	$\leq_K$	$\top$

0	$\leq_B$	$\bot$	$\leq_B$	1
0	$\leq_B$	$\top$	$\leq_B$	1

Our goal is to provide (ξ, $\leq_B$, $\leq_K$, $\neg$) with a bilattice structure. For this, we need to endow (ξ, $\leq_K$) with a lattice structure. Thus we define $\sqcup$ and $\sqcap$ as the meet and the join for the knowledge order:

$\sqcup$	1	0	$\top$	$\bot$
1	1	$\top$	$\top$	1
0	$\top$	0	$\top$	0
$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	1	0	$\top$	$\bot$

$\sqcap$	1	0	$\top$	$\bot$
1	1	$\bot$	1	$\bot$
0	$\bot$	0	0	$\bot$
$\top$	1	0	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

We also want that (ξ, $\leq_B$) fits a lattice structure. Thus, similarly we define $\boxplus$ and $\boxdot$ as meet and join for the Boolean order:

$x \leq_K (x \sqcup y)$	$x \leq_B (x \boxplus y)$	$x \leq_B y$ and $z \leq_B t \Rightarrow x \sqcup z \leq_B y \sqcup t$
$x \leq_K y \Rightarrow (x \sqcup z) \leq_K (y \sqcup z)$	$(x \boxdot y) \leq_B x$	$x \leq_B y$ and $z \leq_B t \Rightarrow x \sqcap z \leq_B y \sqcap t$
$x \leq_K y \Rightarrow (x \sqcap z) \leq_K (y \sqcap z)$	$x \leq_B y \Rightarrow (x \boxdot z) \leq_B (y \boxdot z)$	$x \leq_K y$ and $z \leq_K t \Rightarrow x \boxplus z \leq_K y \boxplus t$
$\bot \leq_K (x \sqcup y) \leq_K \top$	$x \leq_B y \Rightarrow (x \boxdot z) \leq_B (y \boxdot z)$	$x \leq_K y$ and $z \leq_K t \Rightarrow x \boxdot z \leq_K y \boxdot t$
$\bot \leq_K (x \sqcap y) \leq_K \top$	$0 \leq_B (x \boxplus y) \leq_B 1$	
$x \leq_K y \Rightarrow \neg x \leq_K \neg y$	$0 \leq_B (x \boxdot y) \leq_B 1$	
	$x \leq_B y \Rightarrow \neg y \leq_B \neg x$	

Table 2: $\leq_K$ and $\leq_B$ properties

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

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

Finally, we define the $\neg$ operation. Indeed, we expect that it inverts the notion of truth from a Boolean point of view, but its role with respect to $\leq_K$ has to be transparent : we know no more and no less about x and $\neg x$:

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

With these definitions, orders have the properties described in table 2, proved in [**?**].
The structure $(\xi, \leq_B, \leq_K, \neg)$ is a bilattice: (1) By construction, $(\xi, \leq_K)$ is a lattice with $\bot$ and $\top$ as extremums, and so is $(\xi, \leq_B)$ with 0 and 1 as extremums. According to table 2, we can ensure that $\neg$ operator inverts the Boolean order and preserves the knowledge order. Finally, obviously, $\neg\neg x = x$, for each element of ξ.

Although the ξ algebra can be considered as an extension of Boolean algebra, it is not itself a Boolean algebra,because for instance, $x \boxplus \neg x$ is not always equal to 1. Thus, we cannot benefit from Boolean algebra laws. Hence, we study which of these laws hold. Results are detailed in table 3 and the proofs are in [**?**].

These laws are useful to compute the ξ-equation system solutions. Moreover, distributivity is a cornerstone to apply bilattice properties.

3.1.2 The ξ Algebra Encoding

As already said, the goal of the equational semantics is to associate with each program a ξ-equation system computing the status of its events in each instant. To make this computation efficient, we define an encoding of ξ elements as pair of Boolean ones. There exist several possible encoding functions and we choose the most suited one to express the increasing information with respect to the $\leq_K$ order:

$$e : \xi \mapsto \mathbb{B} \times \mathbb{B} : x \in \xi,\ e(x) = (x_h, x_l)$$

`Neutral and absorbing element laws:`
$\bot \sqcup x = x \quad 1 \boxplus x = 1 \quad 0 \boxdot x = 0 \quad \top \sqcup x = \top$
$\bot \sqcap x = \bot \quad 0 \boxplus x = x \quad 1 \boxdot x = x \quad \top \sqcap x = x$
`Distributivity laws:`
$(x \boxplus y) \boxdot z = (x \boxdot z) \boxplus (y \boxdot z) \quad (x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z) \quad (x \sqcup y) \boxplus z = (x \boxplus z) \sqcup (y \boxplus z)$
$(x \boxdot y) \boxplus z = (x \boxplus z) \boxdot (y \boxplus z) \quad (x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z) \quad (x \sqcup y) \boxdot z = (x \boxdot z) \sqcup (y \boxdot z)$
$(x \sqcap y) \boxplus z = (x \boxplus z) \sqcap (y \boxplus z) \quad (x \boxplus y) \sqcup z = x \sqcup z \boxplus y \sqcup z \quad (x \boxdot y) \sqcup z = x \sqcup z \boxdot y \sqcup z$
$(x \sqcap y) \boxdot z = (x \boxdot z) \sqcap (y \boxdot z) \quad (x \boxplus y) \sqcap z = x \sqcap z \boxplus y \sqcap z \quad (x \boxdot y) \sqcap z = x \sqcap z \boxdot y \sqcap z$
`Associativity laws:`
$(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z) \quad (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z) \quad (x \boxplus y) \boxplus z = x \boxplus (y \boxplus z) \quad (x \boxdot y) \boxdot z = x \boxdot (y \boxdot z)$
`Absorption laws:`
$x \sqcup x = x \quad x \sqcap x = x \quad x \boxplus x = x \quad x \boxdot x = x$
`Idempotence laws:`
$x \sqcup x = x \quad x \sqcap x = x \quad x \boxplus x = x \quad x \boxdot x = x$
`De Morgan laws:`
$\neg(x \boxdot y) = \neg x \boxplus \neg y \quad \neg(x \boxplus y) = \neg x \boxdot \neg y \quad \neg(x \sqcup y) = \neg(x) \sqcup \neg(y) \quad \neg(x \sqcap y) = \neg(x) \sqcap \neg(y)$

Table 3: The ξ algebra properties

Here $\mathbb{B}$ is the usual Boolean set with two elements: tt and ff. e is defined as:

$$\begin{array}{lcl} \bot & \mapsto & (ff, ff) \\ 0 & \mapsto & (ff, tt) \\ 1 & \mapsto & (tt, ff) \\ \top & \mapsto & (tt, tt) \end{array}$$

The previously described encoding function extends to the ξ operators. The structure $(\mathbb{B}, \leq)$ is a complete lattice for the $ff \leq tt$ order. The structure: $\mathbb{B} \bigodot \mathbb{B} = (\mathbb{B} \times \mathbb{B}, \leq_B, \leq_K, \neg)$ defined as follows

$$\begin{array}{lll} (x_1, x_2) \leq_B (y_1, y_2) & \text{iff} & x_1 \leq y_1 \textit{ and } y_2 \leq x_2 \\ (x_1, x_2) \leq_K (y_1, y_2) & \text{iff} & x_1 \leq y_1 \textit{ and } x_2 \leq y_2 \\ \neg(x_1, x_2) = (x_2, x_1) & & \end{array}$$

is a bilattice. Then, the following theorem holds:

Theorem 1. $(\xi, \leq_B, \leq_K, \neg)$ and $\mathbb{B} \bigodot \mathbb{B}$ are isomorphic.

To prove this theorem we show that the encoding e previously defined is an isomorphism between $(\xi, \leq_B, \leq_K, \neg)$ and $\mathbb{B} \bigodot \mathbb{B}$. Indeed, the four binary operations and the negation one of the $(\xi, \leq_B, \leq_K, \neg)$ bilattice are preserved in $\mathbb{B} \bigodot \mathbb{B}$. The proof is detailed in [3].

As a consequence of the theorem, we can extend the encoding e previously defined for ξ elements to the bilattice $(\xi, \leq_B, \leq_K, \neg)$ operators (in the following equations, $+$ and $.$ denote the join and meet operations of the lattice $(\mathbb{B}, \leq)$):

$$\left[\begin{array}{lcl} e(x \sqcup y) & = & (x_h + y_h, x_l + y_l) \\ e(x \sqcap y) & = & (x_h . y_h, x_l . y_l) \\ e(x \boxplus y) & = & (x_h + y_h, x_l . y_l) \\ e(x \boxdot y) & = & (x_h . y_h, x_l + y_l) \end{array} \right]$$

Thus, we can efficiently convert ξ-equation systems into the Boolean universe.

3.1.3 Extension to Environments

Owing to the ξ algebra, we can now formally introduce the notion of *environments*. Environments are finite sets of events where each event has a single status.

More formally, consider a finite set of events $\mathscr{S} = \{S_0, S_1, ...S_n, ...\}$. A valuation $\mathscr{V} : \mathscr{S} \mapsto \xi$ is a function that maps an event $S \in \mathscr{S}$ to a status value in ξ. Each valuation $\mathscr{V}$ defines an environment: $E = \{S^x \mid S \in \mathscr{S}, x \in \xi, \mathscr{V}(S) = x\}$. The goal of the semantics is to refine the status of the events of a program in each instant from $\bot$ to $\top$ according to the knowledge order ($\leq_K$).

Then, for each program P, built with ADeL operators, let us denote $\mathscr{S}(P)$ the finite set of its events and $\mathscr{E}(P)$ the set of all possible environments built from $\mathscr{S}(P)$. Operations in (ξ, $\leq_B$, $\leq_K$, $\neg$) can be extended to environments[3]:

$$\begin{array}{lcl} \neg E & = & \{S^x | S^{\neg x} \in E\} \\ E \boxplus E' & = & \{S^z | \exists S^x \in E, \exists S^y \in E', z = x \boxplus y\} \\ & & \cup \{S^x | S^x \in E, \nexists y \in \xi, S^y \in E'\} \\ & & \cup \{S^y | S^y \in E', \nexists x \in \xi, S^x \in E\} \end{array}$$

We introduce an order relation ($\preceq$) on environments as follows:

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

Thus $E \preceq E'$ means that each element of E is less than an element of E' according to the lattice knowledge order of ξ. As a consequence, the $\preceq$ relation is a total order on $\mathscr{E}(p)$ and $\sqcup$and $\sqcap$operations are monotonic according to $\preceq$. Moreover, ($\mathscr{E}(P)$, $\preceq$) is a complete lattice, its greatest element is $\{S^\top \mid S \in \mathscr{S}(P)\}$ and its least element is $\{S^\bot \mid S \in \mathscr{S}(P)\}$. According to Tarski's theorem, each monotonic increasing function F has a least fixed point, computed by iteration of F from the least element [8]. This ensures that the behavioral semantics has solutions.

A specific operation of "temporal translation" (*Pre*) is also needed on environments, it is useful to express the semantics of the temporal operators ADel supplies.

$$Pre(E) = \{S^\bot \mid S^x \in E\} \cup \{S^x_{pre} \mid S^x \in E\}$$

This operation creates new occurrences of events in the environment to record the status of events at the previous instant of the logical time. Each event S^x is renamed as S^x_{pre} and the event $S^\bot$ is added. $S^\bot$ status will be increased by operations on the environment performed by the semantics rules when needed by the temporal operators of ADel.

For environments, the $\sqcup$operation is also called "unification".

3.2 Behavioral Semantics

This semantics formalizes each reaction of a program by formally computing the output environment from the input one. It defines a set of rewriting rules of the form $P \xrightarrow[E]{E'} P'$ which describes the program execution, and where P is an ADeL program, E and E' are respectively the input and output environments, and P' is the derivative of P, which represents the new program that will react to the next input environment. For a program P, an input environment is composed of input events, present events having 1 for status and output events with $\bot$ as status.

The equational semantics runs structurally on a program to compute its equation system. Thus, the rewriting rules of the whole program apply from the root instruction following structurally the syntactic tree of the program. A rule has the form:

$$p \xrightarrow[E]{E', term} p'$$

[3]We introduce only the operations needed to define both semantics. However, the five operators of ξ can be similarly extended.

where p and p' are two instructions of ADeL, E is the input environment, E' is the resulting output environment, and $term$ is a Boolean flag which describes the termination of p, and which turns to true when p terminates.

Operator nothing
Operator nothing doesn't have an influence on the current environment, it starts and ends in the same instant and the flag $term$ is always true.

$$\textbf{nothing} \xrightarrow[E]{E,\, tt} \textbf{nothing}$$

Operator parallel

The operator **parallel** ($\|$) has two argument instructions, it computes them concurrently, possibly broadcasting events between them. Thus the evolution of both instructions can impact both environments. The operator ends when the two instructions terminate, i.e. when $term_1$ and $term_2$ become true and the resulting output environment is the unification of the respective resulting environments computed for p_1 and p_2.

$$\frac{p_1 \xrightarrow[E_1 \sqcup E_2]{E_1, term_{p_1}} p_1' \quad , \quad p_2 \xrightarrow[E \sqcup E_1]{E_2, term_{p_2}} p_2'}{p_1 \| p_2 \xrightarrow[E]{E_1 \sqcup E_2,\ term_{p_1}\ and\ term_{p_2}} p_1' \| p_2'}$$

Operator then
The **then** operator has two arguments. It behaves as a sequencing operator, the computation of the second arguments cannot start until the first argument ends (rule $then1$).

$$\frac{p_1 \xrightarrow[E]{E_1,\, ff} p_1'}{p_1 \textbf{ then } p_2 \xrightarrow[E]{E_1,\, ff} p_1' \textbf{ then } p_2} \quad (then1)$$

When the first argument ends, the environment E evolves and transforms into a new output environment called E_1, which is the input environment to compute the operator output environment. When the second argument terminates, the environment E_1 increases to a new resulting output environment E_2 (rule $then2$).

$$\frac{p_1 \xrightarrow[E]{E_1,\, tt} \textbf{nothing} \quad , \quad p_2 \xrightarrow[E_1]{E_2, term_{p_2}} p_2'}{p_1 \textbf{ then } p_2 \xrightarrow[E_1]{E_2, term_{p_2}} p_2'} \quad (then2)$$

Operator if then else

This operator has the usual behavior of test operators. Its behavior depends on the condition to verify. If the condition evaluates to 1 with respect to the input environment[4], the then statement is executed:

[4] denoted $(cond, E) \rightarrowtail 1$ in the rules

$$\frac{p_1 \xrightarrow[E]{E_1, term_{p_1}} p_1', p_2 \xrightarrow[E]{E_2, term_{p_2}} p_2',\ (cond, E) \rightarrowtail 1}{\textbf{if}(cond, p_1, p_2) \xrightarrow[E]{E_1, term_{p_1}} p_1'} \quad (if1)$$

Otherwise , the else statement is executed:

$$\frac{p_1 \xrightarrow[E]{E_1, term_{p_1}} p_1', p_2 \xrightarrow[E]{E_2, term_{p_2}} p_2', (cond, E) \rightarrowtail 1}{\textbf{if}(cond, p_1, p_2) \xrightarrow[E]{E_2, term_{p_2}} p_2'} \quad (if2)$$

Operator wait *S*

The semantics of "**wait**" provides that it does not react in the first instant. So we introduce an intermediate operator that we call "**iwait**" to express the behavior of wait. Thus, first **wait** *S* is not ready to terminate and leave and is rewritten into **iwait** *S* .

$$\textbf{wait S} \xrightarrow[E]{E, ff} \textbf{iwait S} \quad (wait)$$

iwait *S* reacts instantaneously to the presence of the event *S*. If *S* is present, *term* becomes true and *iwait* terminates. It rewrites int **nothing** and without having any influence on the current environment.

$$\frac{S^1 \in E}{\textbf{iwait S} \xrightarrow[E]{E, tt} \textbf{nothing}}$$

If *S* is not present, the value of *term* is false and the **iwait** *S* does not evolve in that instant, and continues to wait for *S* in the next instants without having any influence on the environment.

$$\frac{S^1 \notin E}{\textbf{iwait S} \xrightarrow[E]{E, ff} \textbf{iwait S}}$$

Operator stop

The behavior of this operator depends on the computation of its body statement *p* and on the presence of the aborting signal *S*. However, the presence or absence of *S* has no influence in the first reaction. So, to express the rules of the **stop** operator, we introduce its immediate version (**istop**), similarly to the **wait** operator. Hence, we wait at least one instant:

$$\frac{p \xrightarrow[E]{E_1, term_p} p'}{\textbf{stop}(p,\ S,\ S_1) \xrightarrow[E]{E_1, term_p} \textbf{istop}(p',\ S,\ S_1)} \quad (stop)$$

Then, we describe the rules for the **istop** operator. If *S* is not present and *p* terminates, then the operator **istop** (*p*, *S*, S_1) ends and rewrites into **nothing**. The environment *E* becomes E_1 as the result of the application of the semantic rules to *p* (rule *istop*1).

$$\frac{p \xrightarrow[E]{E_1,\ tt} \textbf{nothing} \quad , \quad S \notin E}{\textbf{istop}(p,\ S,\ S_1) \xrightarrow[E]{E_1,\ tt} \textbf{nothing}} \quad (istop1)$$

If S is not present p continues its execution. The environment evolves into a new resulting output environment E_1, and the operator waits for S in the next instant(rule $istop2$).

$$\frac{p \xrightarrow[E]{E_1, ff} p' \quad , \quad S \notin E}{\textbf{istop}(p,\ S,\ S_1) \xrightarrow[E]{E_1,\ ff} \textbf{istop}(p',\ S,\ S_1)} \quad (istop2)$$

If S is present, then the computation of the operator waits the end of the current instant of the execution of p and finishes in rewriting into **nothing**. Moreover $term$ becomes true. The final environment is the output environment E_1 resulting from the execution of p, where the status of the event S_1 becomes 1 (rule $istop3$).

$$\frac{p \xrightarrow[E]{E_1, term_p} p' \quad , \quad S \in E}{\textbf{istop}(p,\ S,\ S_1) \xrightarrow[E]{E_1 \cup \{S_1^1\},\ tt} \textbf{nothing}} \quad (istop3)$$

Operator while

The operator **while** behaves like a loop which terminates only when its condition is no more 1. If the condition evaluates to 1 with respect to the status of events in the current environment, the operator **while**($cond$, p) never terminates and behaves like p' **then while**($cond$,p) (rule $while1$).

$$\frac{p \xrightarrow[E]{E_1, term_p} P_1' \ , \ (cond, E) \rightarrowtail 1}{\textbf{while}(cond, p) \xrightarrow[E]{E_1,\ ff} p' \textbf{ then while}(cond, p)} \quad (while1)$$

If the condition evaluates to 0, the operator **while** ends and rewrites into **nothing**, $term$ becomes true. The final environment is the output environment E_1 resulting from the execution of p (rule $while2$).

$$\frac{p \xrightarrow[E]{E_1, term_p} p',\ (cond, E) \rightarrowtail 0}{\textbf{while}(cond, p) \xrightarrow[E]{E_1,\ tt} \textbf{nothing}} \quad (while2)$$

Operator during

The operator **during** has two arguments and its computation depends on the behaviors both of them. The first argument cannot be executed before the second argument starts (rule $during1$).

$$\frac{p_2 \xrightarrow[E]{E_2,\ ff} p_2'}{p_1 \textbf{ during } p_2 \xrightarrow[E]{E_2,\ ff} p_1 \textbf{ during } p_2'} \quad (during1)$$

The second argument cannot finishes before the first one:

$$\frac{\text{not } p_1 \xrightarrow[E]{E_1,\ term_{p_1}} p_1' \quad , \quad p_2 \xrightarrow[E]{E_2,} \textbf{nothing}}{p_1 \textbf{ during } p_2 \xrightarrow[E]{E_2,\ tt} \textbf{nothing}} \quad (during2)$$

$$\frac{p_1 \xrightarrow[E \sqcup E_2]{E_1,\ ff} p_1' \quad , \quad p_2 \xrightarrow[E \sqcup E_1]{E_2,\ tt} \textbf{nothing}}{p_1 \textbf{ during } p_2 \xrightarrow[E]{E_1 \sqcup E_2,\ tt} \textbf{nothing}} \quad (during2-bis)$$

When p_2 started, p_1 can start. At one point, the two arguments will be executed in parallel and both of them will have an impact on both of their environments (rule *during*3).

$$\frac{p_1 \xrightarrow[E\sqcup E_2]{E_1,\, ff} p_1' \quad , \quad p_2 \xrightarrow[E\sqcup E_1]{E_2,\, ff} p_2'}{p_1 \textbf{ during } p_2 \xrightarrow[E]{E_1\sqcup E_2,\, ff} p_1' \textbf{ during } p_2'} \quad (during3)$$

Finally, the second argument cannot terminates before the end of the first argument: when p_1 terminates, it's flag will change to true and it rewrites into **nothing**, and then p_2 can terminate its execution. The final resulting output environment will be the last environment resulting from the execution of p_2 (rule *during*4).

$$\frac{p_1 \xrightarrow[E]{E_1,\, tt} \textbf{nothing} \quad , \quad p_2 \xrightarrow[E]{E_2, term_{p_2}} p_2'}{p_1 \textbf{ during } p_2 \xrightarrow[E]{E_1\sqcup E_2, term_{p_2}} p_2'} \quad (during4)$$

Operator local
The **local** operator behaves as an encapsulation. It is used to define events that are not visible in the surrounding environment.

$$\frac{p \xrightarrow[E\cup\{S^{\perp}\}]{E',\, term_p} p'}{\textbf{local}(S,p) \xrightarrow[E]{E'\setminus\{S\},\, term_p} \textbf{local}(S,p')}$$

Operator timeout
The behavior of this operator p **timeout** $S\{p_1\}$**alert**S_1 depends on the computation of its instruction p and on the status of S. If S is not present and p terminates, p_1 will start its execution.(rule *timeout*1).

$$\frac{P \xrightarrow[E]{E_1,\, tt} \textbf{nothing},\ p_1 \xrightarrow[E1]{E_2, term_{p_1}} p_1'}{\textbf{timeout}(p,\ S\ , p_1, S_1) \xrightarrow[E]{E_2, term_{p_1}} p_1'} \quad (timeout1)$$

If S is present (i.e., timeout elapsed), the computation of the operator stops the execution of p and finishes by generating **nothing** as final result and changing *term* to true. The final environment is the output environment E, where the status of the event S_1 becomes 1(rule *timeout*2).

$$\frac{p \xrightarrow[E]{E_1, term_p} p' \quad , \quad (S,E) \rightarrowtail 1}{\textbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E]{E_1\cup\{S_1^1\},\, tt} \textbf{nothing}} \quad (timeout2)$$

If S is not present in the environment, its status can be either $\perp$ or 0. If it is $\perp$, the evaluation cannot progress and it is terminated (rule *timeout*3):

$$\frac{p \xrightarrow[E]{E_1, term_p} p' \quad , \quad (S,E) \rightarrowtail \perp}{\textbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E]{E_1,\, tt} \textbf{nothing}} \quad (timeout3)$$

Otherwise, if S is absent, p continues its evaluation (rule $timeout4$):

$$\frac{p \xrightarrow[E]{E_1, ff} p' \quad , \quad (S,E) \rightarrowtail 0}{\textbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E]{E_1,\ ff} \textbf{timeout}(p',\ S,\ p_1, S_1)} \quad (timeout4)$$

Operator alert S
The **alert** operator emits an event of alert to the environment, it starts and ends in the same instant and changes the flag *term* to true:

$$\textbf{alert S} \xrightarrow[E]{E \cup \{S^1\},\ tt} \textbf{nothing}$$

3.3 Equational Semantics

The equational semantics allows us to make an incremental compilation of the ADeL programs by translating each program into a ξ-equation system. An equation system is defined as the 3-tuple $< V, R, D >$ where V contains variables representing the status of the input events, the output events, and the local events. R are the registers, i.e specific variables acting as memories to record values useful to compute the next instant, and D the definition of the equation system to calculate the status of each event.

Similarly to the behavioral semantics, we compute the output environment of a program applying the semantics rules to its root instruction. Thus, we define it first for the operators of ADeL and then we extend these definitions to programs. The equational semantics is a function $\mathscr{S}_e$ which computes an output environment from an input one. An input environment contains the input events of the program where present events have 1 for status while the the output events have $\perp$. Moreover, it also contains the registers needed by the program and introduced for some operators, they have the status computed in the previous reaction.

Let p be an ADeL instruction and E an input environment. We denote $\mathscr{D}_p$, its equation system and $\langle p \rangle_E$ the resulting output environment, computed by $\mathscr{S}_e$. It is defined as follows:
$\mathscr{S}_e(p,E) = \langle p \rangle_E$ iff $E \vdash \mathscr{D}_p \hookrightarrow \langle p \rangle_E$. From the event status valuation of E, the equation system $\mathscr{D}(p)$ results in the event status valuation of $\langle p \rangle_E$. To compute output event values and register next values, we rely on the ξ algebra laws detailed in table 3.

Let P be an ADeL program and E a global input environment (i.e., an environment where output and local event variables have $\perp$ as status and the registers have 0). The equational semantics formalizes a reaction of P with respect to E: $(P,E) \longmapsto E'$ iff $\mathscr{S}_e(\beta(P),E) = E'$, $\beta(P)$ being the instruction representing the body of P, i.e, the root instruction of the syntactic tree of P. Thus, we deduce the equation system of an instruction from semantic rules defined for each operators of the language. To express these rules, we add to each operator three specific events: an input event START to start the instruction, an input event KILL to kill the instruction, and an output event FINISH to send the termination information to the enclosing instruction. For each instruction, its START, KILL and FINISH events are added to the input environment.

The operator equation systems are defined using operator semantic rules to compute the status of the FINISH, output, and local events, according to the status of START, KILL, input and local events.

We describe the equational semantics of ADeL operators and we present their rules.

Operator nothing
As we said previously, **nothing** does nothing, its output environment is calculated as following:

$$E \vdash \mathscr{D}_{nothing} \hookrightarrow \langle \textbf{nothing} \rangle_E.$$

This operator's corresponding equation system is simple :[5].

$$\mathscr{D}_{nothing} = \left[\begin{array}{lcl} \text{FINISH} & = & \text{START} \end{array} \right]$$

Operator wait S
wait S is a temporal operator. It does not react in the first instant. To calculate its output environment $\langle wait\ S \rangle_E$ we use the specific "temporal translation" operation *Pre* (see section 3.1.3). The output environment of wait operator is calculated as following:

$$\langle wait\ S \rangle_E = Pre(E') \text{ and } E \vdash \mathscr{D}_{wait\ S} \hookrightarrow E'$$

The corresponding equation system of the operator **wait S** is as follows. It does not terminates instantaneously, thus, we need to create a register to memorize that the operator execution has started (START has been true).

$$\mathscr{D}_{wait} = \left[\begin{array}{lcl} \text{REG}_+ & = & \text{START} \boxdot \neg\, \text{REG} \boxdot \neg\, \text{KILL} \boxplus\ \text{REG} \boxdot \neg\, S \boxdot \neg\, \text{KILL} \\ \text{FINISH} & = & S \boxdot\ \text{REG} \end{array} \right]$$

Operator parallel

The operator **parallel** unifies (operation $\sqcup$) the respective environments of its two operands. The output environment is calculated using the following rule:

$$\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E \vdash \mathscr{D}_{\|} \hookrightarrow \langle p_1 \| p_2 \rangle_E$$

The rule to define $\mathscr{D}_{p_1 \| p_2}$ introduces two registers REG_1 and REG_2 to record the respective status of the FINISH events of the two parallel arguments, since this operator ends when both of its operands have terminated. Note that the operands do not in general terminate in the same instant.

$$\mathscr{D}_{\|} = \left[\begin{array}{lcl} \text{REG}_1^+ & = & \text{REG}_1 \boxdot\ \neg\text{FINISH}_{p_2} \boxdot\ \neg\text{KILL} \boxplus\ \neg\text{REG}_2 \boxdot \\ & & \text{FINISH}_{p_1} \boxdot\ \neg\text{FINISH}_{p_2} \boxdot\ \neg\text{KILL} \\ \text{REG}_2^+ & = & \text{REG}_2 \boxdot\ \neg\text{FINISH}_{p_1} \boxdot\ \neg\text{KILL} \boxplus\ \neg\text{REG}_1 \boxdot \\ & & \neg\text{FINISH}_{p_1} \boxdot\ \text{FINISH}_{p_2} \boxdot\ \neg\text{KILL} \\ \text{START}_{p_1} & = & \text{START} \\ \text{START}_{p_2} & = & \text{START} \\ \text{KILL}_{p_1} & = & \text{KILL} \\ \text{KILL}_{p_2} & = & \text{KILL} \\ \text{FINISH} & = & \text{REG}_1 \boxdot\ \neg\text{REG}_2 \boxdot \text{FINISH}_{p_2} \boxplus \text{REG}_2 \boxdot\ \neg\ \text{REG}_1 \boxdot\ \text{FINISH}_{p_1} \boxplus \\ & & \neg\text{REG}_1 \boxdot\ \neg\text{REG}_2 \boxdot\ \text{FINISH}_{p_1} \boxdot\ \text{FINISH}_{p_2} \end{array} \right]$$

Operator then
The output environment of this operator is computed as following:

$$\langle p_2 \rangle_{\langle p_1 \rangle_E} \vdash \mathscr{D}_{then} \hookrightarrow \langle p_1 \textbf{ then } p_2 \rangle_E$$

[5] In the following, to express the equation systems of an operator, its specific events will be denoted START, KILL and FINISH while the specific events of its potential arguments will be indexed with the argument respective name.

In the operator equation system, the second arguments starts when the first argument finishes (equation 1). The FINISH event of the operator is related to the end of the second argument. (equation 2).

$$\mathscr{D}_{then} = \left[\begin{array}{lcll} \text{START}_{p_1} & = & \text{START} & \\ \text{KILL}_{p_1} & = & \text{KILL} & \\ \text{START}_{p_2} & = & \text{FINISH}_{p_1} & (1) \\ \text{KILL}_{p_2} & = & \text{KILL} & \\ \text{FINISH} & = & \text{FINISH}_{p_2} & (2) \end{array}\right]$$

Operator stop
The output environment of the **stop** instruction is computed as following:

$$\langle p \rangle_E \vdash \mathscr{D}_{stop} \hookrightarrow \langle \textbf{stop}(p, S, S_1) \rangle_E.$$

The operator **stop** does not terminates instantaneously when the aborting event is present. Indeed, in this case, the environment continues to evolve until the current instant terminates.

$$\mathscr{D}_{stop} = \left[\begin{array}{lcl} \text{REG}^+ & = & \text{START} \boxplus (\text{REG} \boxdot \neg S) \boxplus \neg \text{FINISH}_p \\ \text{START}_p & = & \text{START} \\ \text{KILL}_p & = & (S \boxdot \neg\text{KILL} \boxdot \text{REG}) \boxplus \text{KILL} \\ \text{FINISH} & = & \text{REG} \boxdot (\text{FINISH}_p \boxplus S) \\ S_1 & = & \text{REG} \boxdot S \boxdot \neg\text{FINISH}_p \end{array}\right]$$

Operator while
The output environment of this operator is computed as following:

$$\langle p \rangle_E \vdash \mathscr{D}_{while} \hookrightarrow \langle \textbf{while}(p, cond) \rangle_E.$$

The while statement terminates only when the condition event is no more present.

$$\mathscr{D}_{while} = \left[\begin{array}{lcl} \text{REG}^+ & = & (\text{REG} \boxdot cond) \boxplus (\neg\text{REG} \boxdot cond \boxdot \text{START} \boxdot \neg\text{KILL}) \\ \text{START}_p & = & (\text{REG} \boxdot cond \boxdot \text{FINISH}_p) \boxplus (\neg\text{REG} \boxdot cond \boxdot \text{START} \boxdot \neg\text{KILL}) \\ \text{KILL}_p & = & (\text{REG} \boxdot \neg cond) \boxplus (\neg\text{REG} \boxdot \text{KILL}) \\ \text{FINISH} & = & (\text{REG} \boxdot \neg cond) \boxplus (\neg\text{REG} \boxdot \neg cond \boxdot \text{START} \boxdot \neg\text{KILL}) \end{array}\right]$$

Operator if then else
The output environment of this operator is calculated as following:

$$\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E \vdash \mathscr{D}_{if} \hookrightarrow \langle \textbf{if}(cond, p_1, p_2) \rangle_E.$$

In the equation system of this operator, the START event is linked to the "then statement p_1" when the test event is present and to the "else statement p_2" when it is not. The FINISH event of this operator is present when the FINISH event of its selected argument is present.

$$\mathscr{D}_{if} = \left[\begin{array}{lcl} \text{START}_{p_1} & = & \text{START} \boxdot cond \\ \text{START}_{p_2} & = & \text{START} \boxdot \neg cond \\ \text{KILL}_{p_1} & = & \text{KILL} \\ \text{KILL}_{p_2} & = & \text{KILL} \\ \text{FINISH} & = & \text{FINISH}_{p_1} \boxplus \text{FINISH}_{p_2} \end{array}\right]$$

Operator during

The equations of this operator reflects the fact that the first argument starts after the second one and terminates before the end of the second argument. Same to parallel, this operator unifies (operation $\sqcup$) the input environments of its two operands. The output environment is calculated using the following rule:

$$\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E \vdash \mathscr{D}_{during} \hookrightarrow \langle p_1 \textbf{ during } p_2 \rangle_E$$

We need two registers to record the state of the two arguments. Its corresponding equation system is the following:

$$\mathscr{D}_{during} = \left[\begin{array}{lcl} \mathrm{REG}_1^+ & = & (\mathrm{REG}_1 \boxdot \mathrm{REG}_2 \boxdot \neg\mathrm{FINISH}_{p_2}) \boxplus (\neg\mathrm{REG}_1 \boxdot \neg\mathrm{REG}_2 \boxdot \mathrm{START}) \boxplus \\ & & (\mathrm{REG}_1 \boxdot \neg\mathrm{REG}_2 \boxdot \neg\mathrm{FINISH}_{p_1} \boxdot \neg\mathrm{FINISH}_{p_2}) \\ \mathrm{REG}_2^+ & = & (\mathrm{REG}_1 \boxdot \neg\mathrm{REG}_2 \boxdot \mathrm{FINISH}_{p_1} \boxdot \neg\mathrm{FINISH}_{p_2}) \boxplus \\ & & (\neg\mathrm{REG}_1 \boxdot \mathrm{REG}_2 \boxdot \neg\mathrm{FINISH}_{p_2}) \boxplus (\neg\mathrm{REG}_1 \boxdot \neg\mathrm{REG}_2 \boxdot \mathrm{START}) \\ \mathrm{START}_{p_2} & = & \neg\mathrm{REG}_1 \boxdot \neg\mathrm{REG}_2 \boxdot \mathrm{START} \\ \mathrm{START}_{p_1} & = & \mathrm{REG}_1 \boxdot \mathrm{REG}_2 \boxdot \neg\mathrm{FINISH}_{p_2} \\ \mathrm{KILL}_{p_1} & = & \mathrm{KILL} \\ \mathrm{KILL}_{p_2} & = & \mathrm{KILL} \\ \mathrm{FINISH} & = & (\neg\mathrm{REG}_1 \boxdot \mathrm{REG}_2 \boxdot \mathrm{FINISH}_{p_2}) \boxplus (\mathrm{REG}_1 \boxdot \mathrm{REG}_2 \boxdot \mathrm{FINISH}_{p_2}) \\ & & (\mathrm{REG}_1 \boxdot \neg\mathrm{REG}_2 \boxdot \mathrm{FINISH}_{p_1} \boxdot \mathrm{FINISH}_{p_2}) \end{array} \right]$$

Operator local
We consider that the events declared local are new and do not belong to the input environment. The output environment of this operator is calculated as following:

$$\langle p \rangle_E \vdash \mathscr{D}_{local} \hookrightarrow \langle \textbf{local}(p,S) \rangle_E$$

The body statement p starts when the operator local starts it's execution, and the end of p causes the end of the local operator. The equation system is as follows :

$$\mathscr{D}_{local} = \left[\begin{array}{lcl} \mathrm{START}_p & = & \mathrm{START} \\ \mathrm{KILL}_p & = & \mathrm{KILL} \\ \mathrm{FINISH} & = & \mathrm{FINISH}_p \end{array} \right]$$

Operator timeout
The output environment of instruction p **timeout** S $\{p_1\}$ **alert** S_1 is computed as follows:

$$\langle p_1 \rangle_{\langle p \rangle_E} \vdash \mathscr{D}_{timeout} \hookrightarrow \langle \textbf{timeout}(p,S,p_1,S_1) \rangle_E .$$

The $\mathscr{D}_{timeout(p,S,p_1)}$ equation system contains also two registers to memorize the way this instruction terminates: either with the normal termination of its argument (p) or when the timeout signal becomes true. To express the rule for timeout operator, we use the same rules to denote events as in the previous operator.

$$\mathscr{D}_{timeout} = \left[\begin{array}{lcl}
\text{REG}_1^+ & = & \text{REG}_1 \boxdot \neg S \boxdot \neg\text{FINISH}_p \boxdot \neg\text{KILL} \boxplus \\
 & & \neg\text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{START} \boxdot \neg\text{KILL} \\
\text{REG}_2^+ & = & \text{REG}_1 \boxdot \text{REG}_2 \boxdot \neg\text{FINISH}_{p_1} \boxdot \neg\text{KILL} \boxplus \\
 & & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \neg S \boxdot \neg\text{FINISH}_p \boxdot \neg\text{FINISH}_{p_1} \boxdot \neg\text{KILL} \boxplus \\
 & & \neg\text{REG}_1 \boxdot \text{REG}_2 \boxdot \neg\text{FINISH}_{p_1} \\
\text{START}_p & = & \neg\text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{START} \\
\text{START}_{p_1} & = & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \neg S \boxdot \text{FINISH}_p \\
\text{KILL}_p & = & \text{KILL} \\
\text{KILL}_{p_1} & = & S \boxdot \neg\text{KILL} \boxdot \text{REG}_1 \boxplus \text{KILL} \\
\text{FINISH} & = & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot S \boxplus \neg\text{REG}_1 \boxdot \text{REG}_2 \boxdot \text{FINISH}_{p_1} \boxplus \\
 & & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \neg S \boxdot \text{FINISH}_p \boxdot \neg\text{FINISH}_{p_1} \\
S_1 & = & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot S \boxdot \neg\text{FINISH}_p
\end{array}\right]$$

Operator alert S
This operator sends immediately an event of alert. The output environment is calculated using the following rule:

$$E \vdash \mathscr{D}_{alert} \hookrightarrow \langle \textbf{alert S} \rangle_E$$

The equation system of this operator is simple:

$$\mathscr{D}_{alert} = \left[\begin{array}{lcl} \text{FINISH} & = & \text{START} \\ S & = & \text{FINISH} \end{array}\right]$$

3.4 Relation between Behavioral and Equational Semantics

The behavioral semantics is a "macro" step semantics that gives the meaning of a reaction for each ADeL program. Nevertheless, a reaction is the least fixed point of a "micro" step semantics [1] that computes the output environment from the initial one. According to the fact that the $\sqcup$ operation is monotonic with respect to the $\preceq$ order on environments, we can ensure that for each instruction, this least fixed point exists. Practically, we have $p \xrightarrow[E]{E',term} p'$ if there is a sequence of micro steps:

$$p \xrightarrow[E]{E_1,term1} p_1, p_1 \xrightarrow[E_1]{E_2,term2} p_2, ..., p_n \xrightarrow[E_n]{E_{n+1},term_{n+1}} p'$$

At each step $E_{i+1} = F^i(E_i)$. Since the F_i functions rely on the $\boxplus$ operator to increase information about events, they are monotonic and then $\forall i, E_{i+1} \leq_K F^i(E_i)$[6]. Then, we have $E' = \sqcup_n F^n(E_n)$, thus it turns out that E' is the least fixed point of the family of F^n functions. Since $(\xi, \leq_K)$ is a complete lattice, so is the set of environments, and, as a consequence, such a least fixed point exists.

On the other hand, the equational semantics allows us to compile the language by associating an equation system to each operator. To verify that the equational semantics is correct, we prove that it is equivalent to the behavioral one. Indeed, for a program P, we have to show that both semantics agree on the set of emitted events as well as on the termination flag value. To this aim, we prove the following theorem :

[6] We also call $\leq_K$ the extension of the $\leq_K$ order to environments.

Theorem 2. Let p be an ADeL instruction and E its input environment. If $\langle p \rangle_E$ is the resulting environment computed by the equational semantics, then the following property holds:

$$\exists p' \text{ such that } p \xrightarrow[E|_{\mathscr{B}}]{E', e(\text{FINISH}_p)_h} p' \text{ and } \forall o \in E', o \text{ output event, } o \text{ has the same status in } \langle p \rangle_E \text{ and } E'$$

$E\,|_{\mathscr{B}}$ is the environment E where all the START, FINISH, textsckill and REG events have been suppressed. It is composed of the real input, output and local events of an instruction.

The theorem means that if the equational semantics yields a solution, there exists also a behavioral solution with the same outputs. Thus, if the equational semantics computes a termination value (i.e the status of FINISH value of the instruction), the behavioral semantics agrees. However, the status of the FINISH event is a value in ξ while the termination flag of the behavioral semantics is a Boolean. It is why we consider the first projection of the encoding of FINISH in the theorem. Indeed, the relevant values of the FINISH events is either 0 or 1. $\top$ would means that a FINISH event has 2 incompatible status in 2 branches f a parallel operator, this is not possible since the FINISH events are generated for each operators and cannot be shared between 2 branches of a parallel. $\bot$ would means that the START event of the considered operator has never been present and that the execution has never started. Thus, according to the definition of the encoding function (see section 3.1.2), $e(0) = (ff, tt)$ and $e(1) = (tt, ff)$. Then the first projection of 0 encoding is ff and the first projection f 1 encoding is tt.

To prove this theorem, we introduce the notion of size of an instruction (roughly speaking the number of nodes in its syntax tree) and we use it in a proof by induction on the size of a program.

- $[\textbf{nothing}] = 1$
- $[\textbf{wait}S] = 1$
- $[\textbf{alert}S] = 1$
- $[p_1 \parallel p_2] = max([p_1], [p_2]) + 1$
- $[p_1 \textbf{ then } p_2] = [p_1] + [p_2] + 1$
- $[\textbf{stop } p \textbf{ when } S \textbf{ alert}(S_1)] = [p] + 1$
- $[\textbf{while } cond\{p\}] = [p] + 1$
- $[\textbf{if } cond \textbf{ then } p_1 \textbf{ else } p_2] = max([p_1], [p_2]) + 1$
- $[p_1 \textbf{ during } p_2] = [p_1] + [p_2] + 1$
- $[p \textbf{ timeout } S\ \{p_1\}\textbf{alert } S_1] = [p_1] + 1$

The proof is an induction n the size of the ADeL operators.

$[p]$ = 1

First, we prove the theorem for basic statements with size 1, which are **nothing**, **wait** and **alert**. The goal is to prove that in each reaction, both the equational and the behavioral semantics agree with the termination of the operator and compute the same values for output events. In the equational semantics, we distinguish the first reaction where the status of START is 1 from the following where it is 0.

nothing
According to the equational semantics of this operator we have:

In the initial reaction we have: FINISH = START = 1 and thus $e(\text{FINISH})_h = tt$. On the other side we have $term_p = tt$.

We also have for output environment of the behavioral semantics $E' = E\ |_{\mathscr{B}} = \emptyset$ from the operator rule, so the property for output environment holds.

wait *S*

In the initial reaction, we have in the equational semantics REG = 0; KILL = 0; FINISH = 0. On the other part, $term_{\textbf{wait}} = ff$.
Considering the output events of the environment resulting of the application of the rule *wait* of the behavioral semantics, it is clear that there is no change, then the property is true.

When it isn't the initial reaction, according to to the system equation, it is obvious that $\text{REG}_+ = 1$ in the first reaction, so REG = 1 in the current reaction. The proof depends on the status of *S* and falls into two cases:

1. *S is present*
   When *S* is present we have $S = 1$ and we have REG = 1 because it memorizes the value of the previous instant, thus we have FINISH = 1. In the behavioral semantics, it is the first rule of the **iwait** operator which applies and so $term_{iwait} = tt$. On the other side, similarly to the first reaction, we have no change concerning output in the resulting environment of the behavioral semantics.

2. *S is not present*: we have $S = \perp$ or $S = 0$, and REG = 1 as it contains the value of the previous instant. Hence we have FINISH = 0. The second rule of the behavioral semantics for **iwait** applies and then $term_{iwait} = ff$. Moreover, the resulting environment for the behavioral semantics does not change any more.

alert *S*
This operator is instantaneous and we have FINISH = START = 1. Concerning the behavioral semantics, the rule says that $term_{alert} = tt$. Moreover, both semantics add S^1 in their respective output environments and it is the only change. As we start to compute with the same values for the output events in the respective input environments of each semantics, $\forall o \in E'$, *o* has the same status in $\langle \textbf{alert} \rangle_E$.

$[p]$ **= n**

$p_1 \parallel p_2$

With this operator the proof of the theorem falls into 4 cases:

1. p_1 *terminates before* p_2 *does*
   In this case, $\text{FINISH}_{p_1} = 1$. Notice that it is the first reaction where $\text{FINISH}_{p_1} = 1$, since the FINISH events are 1 only once, according to equation systems. Thus, the REG_1 register was 0 and will become 1 in the next reaction. For the current reaction,REG_1 = REG_2 = 0 and then FINISH = 0. In the behavioral semantics, we have $term_{p_1} = tt$, $term_{p_2} = ff$ thus $term_{\parallel} = ff$. Notice that, the

application of the behavioral semantics rule is:

$$\frac{p_1 \xrightarrow[E \sqcup E_2]{E_1,1} \textbf{nothing} \quad , \quad p_2 \xrightarrow[E \sqcup E_1]{E_2,term_{p_2}} p_2'}{p_1 \| p_2 \xrightarrow[E]{E_1 \sqcup E_2,\ term_{p_2}} \textbf{nothing} \| p_2'}$$

In the next instants, while FINISH_{p_2} remains 0, REG_1 is 1 and REG_2 is 0, so FINISH is 0 and $term_{\|}$ is ff. When FINISH_{p_2} becomes 1, REG_1 is 1, according to the equation system, but REG_1+ is set to 0, thus REG_1 will become 0 in the next reactions. Moreover, REG_2 is also 0 and then FINISH $= 1$. In this case, taking account the previous comment, we have the following rewriting in the behavioral semantics:

$$\frac{\textbf{nothing} \xrightarrow[E]{E,1} \textbf{nothing} \quad , \quad p_2 \xrightarrow[E]{E_2,1} \textbf{nothing}}{\textbf{nothing} \| p_2 \xrightarrow[E]{E \sqcup E_2,\ 1} \textbf{nothing} \| \textbf{nothing}}$$

because, by induction we know that $term_{p_2} = e(\text{FINISH}_{p_2})_h = tt$.

2. *p_2 terminates before p_1 does*
   This proof for this case is symmetric to the first item.

3. *p_1 and p_2 terminates simultaneously*
   we have REG_1 and REG_2 both equal to 0, thus the equation for FINISH computes 1 and the behavioral semantics rule is the Boolean and of $term_{p_1}$ and $term_{p_2}$. Both of them are tt, so is $term_{\|}$.

4. *p_1 and p_2 never terminates*
   REG_1 and REG_2 remains 0, hence FINISH too. By induction, $term_{p_1}$ and $term_{p_2}$ are ff, thus $term_{\|} = ff$.

Concerning the output event values, by induction we know that all the output events of E_1 have the same status in $\langle p_1 \rangle_E$ and similarly for E_2. In the behavioral semantics, we perform a $\sqcup$ operation on environment. This means that, $\forall o \in E'$, o output event, either $o \in E_1$ and $o \in E_2$ and then, in the resulting environment E', its status is the unification of its respective status in E_1 and E_2. By induction , we know that the status of o in E_1 is the same in $\langle p_1 \rangle_E$ and similarly for its status in E_2. Thus, according to the equational semantics rule, o has the same status in $\langle p_1 \parallel p_2 \rangle_E$. Or $o \in E_1$ and $o \notin E_2$, then the result straightly comes from the induction hypothesis. The dual case is similar.

p_1 **then** *p_2*
The execution of the sequence operator depends on the status of FINISH_{p_1}, hence, the proof falls into two cases:

1. $\text{FINISH}_{p_1} = 0$:
   in the equational semantics, we have $\text{START}_{p_2} = \text{FINISH}_{p_1} = 0$ and FINISH $= \text{FINISH}_{p_2} = 0$. However, as START_{p_2} is 0, FINISH_{p_2} is also 0. In the operator equations, there is no way to have FINISH equal to 1 if START = 0 (A similar induction will provide us with the proof), thus FINISH $= 0$. We know by induction that $term_{p_1} = e(\text{FINISH}_{p_1})_h = 0$. Then, the rule *then*1 of the behavioral semantics is applied, and $term = ff$.
   For output events of E', in this case, by induction, we know that they have the same status in $\langle p_1 \rangle_E$. As $\text{START}_{p_2} = 0$, no equation in p_2 equation system can change the value of p_2 events, so $\langle p_2 \rangle_{\langle p_1 \rangle_E}$ = $\langle p_1 \rangle_E$, and the property holds applying the induction hypothesis.

2. $\text{FINISH}_{p_1} = 1$:
   Here we have by induction $e(\text{FINISH}_{p_2})_h = term_{p_2}$. From the equational semantics, we have $\text{START}_{p_2} = \text{FINISH}_{p_1}$ and $\text{FINISH} = \text{FINISH}_{p_2}$.
   Let o be an output event in $E' = E_2$ in this case. By induction, we know that o has the same status in E_1 and in $\langle p_1 \rangle_E$. However, $E_1 = (\langle p_1 \rangle_E)_{\mathscr{B}}$, hence applying the hypothesis induction for p_2 with $\langle p_1 \rangle_E$ as input event, we get the expected result concerning the output events of E'.

stop p **when** S **alert** S_1

In the initial reaction, we have in the equational semantics $\text{REG} = 0$, $\text{KILL} = 0$ and $\text{FINISH} = 0$. It is the rule *stop* that applies in the behavioral semantics and thus $term_{stop} = ff$.
Moreover, neither the equations or the rule *stop* change the values of the output event.
Notice that, in this first reaction, the equational semantics set $\text{REG}_+ = 1$, so in the next reactions, the value of *reg* will be 1.

For the following reactions, we have three cases:

1. *S is not present and* $\text{FINISH}_p = 1$
   As already said, the value of *reg* is 1. So according to the equation for FINISH, we deduce that its value is 1. In the behavioral semantics, the rule $istop1$ is applied and $term_{stop} = ff$.
   In the $istop1$ rule, the output environment is the output environment of p. The equations associated with the **stop** operator does not change the values of the output events different from FINISH or the next values of the register *reg*. Thus, only the equations associated with p change the values of these output events in the input environment, and by induction, we know that these output event values are the same.

2. *S is not present and* $\text{FINISH}_p \neq 1$
   When S is not present we have $\text{REG} = 1$ but $S = 0$ or $S = \perp$ and $\text{FINISH}_p = 0$ or $\text{FINISH}_p = \perp$, thus $\text{FINISH} = 0$ or $\text{FINISH} = \perp$. However $e(\text{FINISH}_p)_h = ff$ and by induction, we know that it is the value of $term_p$. Thus, the rule $istop2$ of the behavioral semantics is applied, and the value of $term_p = ff = e(\text{FINISH})_h$.
   Concerning the value of the output events in the resulting environment of the behavioral semantics, the same reasoning as for item 1 holds.

3. *S is present*
   When S is present we have $\text{REG} = 1$ and $S = 1$ thus we have $\text{FINISH} = 1$. On the other hand, it is the rule $istop3$ of the behavioral semantics that applies, and so $term_{stop} = tt$.
   Concerning the value of the output events in the resulting environment of the behavioral semantics, in both semantics, the only change is to add S_1^1 to the environment. Thus, relying on the induction hypothesis, we get the expected result.

while $cond\{p\}$

The process of the operator **while** depends on the presence and the absence of the condition, and here the theorem falls into two cases:

1. *cond =1*:
   When the condition is 1 we have in the equational semantics $\text{FINISH} = 0$, whatever the value of *reg* is. Hence, when *cond* is 1, the rule $while1$ of the behavioral semantics applies and then $term_{while} = ff$.

2. $cond = 0$:
   When the condition is 0, if REG $= 1$ then FINISH $= 1$. Otherwise, if $reg = 0$, this means that we are in the first reaction, so START = 1 and then FINISH is also 1. In this case, the rule $while2$ applies and $term_{while} = tt$.

Notice that, if $cond = \bot$, the register remains 0 as START_p and FINISH and there is no rule in the behavioral semantics.

For output event in the output environment of the behavioral semantics (E_1), it is the output environment of the application of the rules for p. By induction we know that its output events have the same status than in $\langle p \rangle_E$, and the equation for the **while** operator do not change the value of output events in the environment. Only the equations of p have an influence on them, so applying the induction hypothesis the property holds.

If *cond* **then** p_1 **else** p_2
The theorem falls into two cases:

1. $cond = 1$
   We have $\text{START}_{p_1} = 1$ *and* $\text{START}_{p_2} = o$ in the equational semantics, and so we have FINISH $= \text{FINISH}_{p_1}$. In the behavioral semantics, the rule $if1$ applies and then the termination flag is $term_{p_1}$. By induction, we also know that $e(\text{FINISH}_{p_1})_h = term_{p_1}$ so $e(\text{FINISH})_h = term_{p_1}$.
   Concerning the output event status of the behavioral semantics (E_1), we know that the output event of E_1 have the same status in $\langle p_1 \rangle_E$. The equational semantics performs the unification of the respective output environment of p_1 and p_2 ($\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E$). However, let us consider o an output belonging to E_1, with a status s_1, it has the same status in $\langle p_1 \rangle_E$. If o is emitted in p_2 by an **alert**, or a **stop** or an **timeout**, its equation in $\langle p_2 \rangle_E$ depends on the FINISH of the operator. So, the status of o in $\langle p_2 \rangle_E$ can be 1 only if this FINISH is 1 otherwise it remains to $\bot$. As, $\text{START}_{p_2} = 0$, FINISH_{p2} cannot be 1 and o will never be set to 1. So, o has status $s_1 \sqcup \bot$ in $\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E$ and according to the definition of the $\sqcup$operation, it has the status s_1. None of the equations of $\mathscr{D}\rangle\{$ can change its status and then o has status s_1 in E', the output environment of the equational semantics.

   • ***condition is*** 0
   If the condition is 0, we have $\text{START}_{p_2} = 1$ and $\text{START}_{p_1} = 0$ in the equational semantics, and so FINISH $= \text{FINISH}_{p_2}$. In the behavioral semantics, the rule $if2$ applies and then the termination flag is $term_{p_2}$. By induction we have $\text{FINISH}_{p_2} = term_{p2}$ so FINISH $= term_{p_2}$.
   Concerning the output event status of the behavioral semantics (E_1), we are in the "dual" case and a similar reasoning leads us to the expected conclusion.

p_1 **during** p_2
The behavior of this operator depends on the respective START and FINISH of p_1 and p_2.

$\mathscr{D}_{during}$ has 2 registers, this means that there are 4 combinations as values for these registers and combinations represent the states of the finite state machine encoded by the equation system.

1. the initial state corresponds to $\text{REG}_1 = 0$ and $\text{REG}_2 = 0$:
   In this first reaction, START $= 1$ and thus, FINISH $= 0$. Moreover, REG_1^+ and REG_2^+ are set to 1. In the behavioral semantics, the rule $during1$ applies and thus $term_{during} = tt = e(\text{FINISH})_h$.

2. the next state is characterized by $\text{REG}_1 = 1$ and $\text{REG}_2 = 1$:
   If $\text{FINISH}_{p_2} = 0$ then $\text{START}_{p_1} = 1$. Moreover, $\text{REG}_1^+ = 1$ and $\text{REG}_2^+ = 0$. With these values for register, the equation for FINISH computes 0 as value. On the other hand, in the behavioral semantics, the rule $during3$ applies, since both p_1 and p_2 continues their evaluation, and in this case

$term_{during} = ff$. If $\text{FINISH}_{p_2} = 1$, $\text{START}_{p_1} = 0$. The next values for registers are $\text{REG}_1^+ = 0$ and $\text{REG}_2^+ = 0$. This means that, the next state is the initial state and that the execution is terminated. The value of FINISH is 1. In this case, the rule *during*2 of the behavioral semantics applies since p_2 finishes and p_1 has not started and then $term_{during} = ff = e(\text{FINISH})_h$.

3. $\text{REG}_1 = 1$ and $\text{REG}_2 = 0$:
   In this state, p_1 and p_2 have begin their evaluation, since START_{p_1} and START_{p_2} have been both 1 in the previous states, but not in the same states (see previous items)). Then, 3 cases appear depending of the respective values of the FINISH event of p_1 and p_2:

   (a) $\text{FINISH}_{p_1} = 0$ and $\text{FINISH}_{p_2} = 0$. Then $\text{FINISH} = 0$. In the behavioral semantics, the rule *during*3 applies and then $term_{during} = ff$. The next values of register are: $\text{REG}_1^+ = 1$ and $\text{REG}_2^+ = 0$, so the next state is unchanged.

   (b) $\text{FINISH}_{p_1} = 0$ and $\text{FINISH}_{p_2} = 1$. Then $\text{FINISH} = 0$. We are in the case where p_1 is executed and does not terminates whereas p_2 terminates. So, in the behavioral semantics, the rule $during2 - bis$ applies and $term_{during} = ff$. In this case, the next state is (0, 0) (next register values), meaning that the evaluation is ended.

   (c) $\text{FINISH}_{p_1} = 1$ and $\text{FINISH}_{p_2} = 0$. In such a situation, $\text{FINISH} = 0$. In the behavioral semantics, it is the rule *during*4 that applies, so $term_{during} = term_{p_2}$. By induction, we know that $term_{p_2} = e(\text{FINISH}_{p_2})_h = ff$. Computing the next values for registers, we can deduce that the next state is 0, 1 (studied in item 4).

   (d) $\text{FINISH}_{p_1} = 1$ and $\text{FINISH}_{p_2} = 1$. In this case, the value of FINISH is 1. On the behavioral semantics parts, it is *during*4 rule that applies since p_1 terminates. So, as for the previous situation, $term_{during} = term_{p_2} = e(\text{FINISH}_{p_2})_h = tt$. In this case, the next state is 0, 0 meaning that the execution is over.

4. $\text{REG}_1 = 0$ and $\text{REG}_2 = 1$:
   We reach this state from state (1, 0) and then p_1 has terminated its execution (see item (c)). Only , the value of FINISH_{p_2} is relevant in this state. With $\text{REG}_1 = 0$ and $\text{REG}_2 = 1$, the equation for FINISH defines it as FINISH_{p_2}. In the behavioral semantics, it is the rule *during*4 that applies, because as p_1 has been rewritten in **nothing**, the application of this rule yields to:

$$\frac{\textbf{nothing} \xrightarrow[E]{E\ tt} \textbf{nothing} \quad , \quad p_2 \xrightarrow[E]{E_2, term_{p_2}} p_2'}{\textbf{nothing during } p_2 \xrightarrow[E]{E_2, term_{p_2}} p_2'}$$

   By induction we know that $term_{p_2} = e(\text{FINISH}_{p_2})_h$ and so $term_{during} = e(\text{FINISH})_h$.

Concerning the output event values in the output environment (E') of the behavioral semantics, let us consider o such an event.

- If the rule *during*1 has been applied, then $E' = E_2$, the output environment of the rewriting of p_2. In the equational semantics, the starting environment to apply equation from $\mathscr{D}_{during}$ is $\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E$. On the other hand, we are in state (0, 0) (because the rule *during*1 is applied), thus $\text{START}_{p_1} = 0$ and no equation in $\mathscr{D}_{p_1}$ are activated, then $\langle p_1 \rangle_E = E$. Moreover, the equations help us to refine the status of events from $\bot$ to $\top$ according to the $\leq_K$ order. Hence, $E \sqcup eqsemp_2E = eqsemp_2E$. By induction, we know that o has the same status in E_2 and $eqsemp_2E$.

- If the rule *during*2 has been applied, then $E' = E_2$. According to the first part of the proof for this operator, in the equational semantics, we are in state (1, 1) and $\text{START}_{p_1} = 0$. Then we can rely on a similar reasoning than in the previous item to conclude.

- When one of the other operator rules is applied, then $E' = E_1 \sqcup E_2$. In the equational semantics, the starting environment to compute $\mathscr{D}_d uring$ equations is $\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E$. By induction, we know that o has the same status in E_1 and $\langle p_1 \rangle_E$ on one hand, and in E_2 and $\langle p_2 \rangle_E$ on the other hand. Thus o has the same status in $E_1 \sqcup E_2$ and in $\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E$. Indeed, none of $\mathscr{D}_d uring$ equations change the value of o status, so o has the same status in $\langle p_1 \textbf{ during } p_2 \rangle_E$.

local(S) **{p}**
In the equational semantics we have FINISH $=$ FINISH$_p$ and we know by induction that FINISH$_p = term_p$ and thus we have FINISH $= term_p$
According to the induction hypothesis, we also know that each output event o in the output environment (E') of the behavioral semantics of p has the same status in $\langle p \rangle_E$. As, the output environment of the operator is $E' \setminus S$ on one hand, and $\langle local \rangle_E \setminus S$ on the other hand, o has the same status in both semantics output environments.

p **timeout** S $\{p_1\}$ **alert** S_1
The operator equation system has two registers, thus there are 4 combinations of values to determine the states.

1. REG$_1 = 0$ and REG$_2 = 0$. By definition, it is the starting state of any evaluation. In this state, START $= 1$ and thus, REG$_1^+ = 1$ and REG$_2^+ = 0$, that determines the next state. Then START$_p$ is 1 and FINISH $= 0$. In the behavioral semantics, it is the rule $timeout3$ which is applied in such a setting. Thus the termination flag is $ff = e(\text{FINISH})_h$.
   The output environment of the rule $timeout4$ is $E_1 \cup \{S_1^0\}$, E_1 being the output environment of the rewriting of p. By induction, we know that each output event in E_1 as the same status in $\langle p \rangle_E$. It is the starting environment to compute the equations in $\mathscr{D}_t imeout$ and the only equation that change the status value of an output is the setting of S_1, which is set to 0. So, each output event in the output environment of the rule $timeout3$ has the same status in $\langle timeout \rangle_E$.

2. REG$_1 = 1$ and REG$_2 = 0$. In this state, we must distinguish the different values for S and FINISH$_p$:

   (a) FINISH$_p = 0$. As S is a 4-valued event, the status of which is refined from $\bot$, we studied the different values it can have:

   - $S = \bot$: with this valuation for registers, FINISH $= 1$. In the behavioral semantics, it is the rule $timeout3$ which applies, and the termination flag is tt.
   - $S = 0$: in this case, FINISH $= 0$. In the behavioral semantics, the rule $timeout4$ applies and the termination flag is ff.
   - $S = 1$: with this valuation for registers, FINISH $= 1$ and its is the rule $timeout2$ of the behavioral semantics that applies. Hence, the termination flag is tt.

   Concerning the output event statuses computed in the behavioral semantics, by induction, we know that each output event in E_1 as the same status in $\langle p \rangle_E$. The value of S_1 is only changed in the last case, but it is set to 1 in the behavioral semantics as well as in the equational one, so the property holds.

   (b) FINISH$_p = 1$. In this case, START$_{p_1}$ is set to 1 by the equations. Similarly to the previous case, we study the different values for S status:

   - $S = \bot$: with this valuation for registers, we also have FINISH $= 1$. In the behavioral semantics, it is the rule $timeout3$ which applies, and the termination flag is tt.

- $S = 0$: in this case, the status of FINISH depends on the status of FINISH_{p_1}. Computing the equation, we get $\text{FINISH} = \text{FINISH}_{p_1}$. In this case, the rule $timeout1$ of the behavioral semantics applies, and the termination flag is equal to $term_{p_1}$. By induction, we have $e(\text{FINISH}_{p_1})_h = term_{p_1}$. So, the result holds.
- $S = 1$: with this valuation for registers, $\text{FINISH} = 1$ and its is the rule $timeout2$ of the behavioral semantics that applies. Hence, the termination flag is tt.

Concerning the output event statuses of the behavioral semantics, the only difference with the case where $\text{FINISH}_p = 0$ is when it is the rule $timeout1$ is applied. Then, in this case, both semantics set the status of S_1 to 0. So, by induction we get the result.

3. $\text{REG}_1 = 0$ and $\text{REG}_2 = 1$. From the possible values for REG_1^+ and REG_2^+ computed in $\mathscr{D}_{timeout}$, we can reach this state only from the state(1, 0). With this valuation for registers, in the equational semantics, we compute $\text{FINISH} = \text{FINISH}_{p_1}$. On the other part, in the behavioral semantics, it is the rule $timeout1$ that continues its application, the termination flag is $term_{p_1}$ and applying the induction hypothesis, we have $e(\text{FINISH})_h = term_{timeout}$.
Concerning the output event values in the respective output environment of both semantics, as neither the behavioral semantics or the equational one change the status of S_1, the result comes from the induction hypothesis.

■

As a corollary, this theorem extends to ADeL programs.

3.5 Compilation and Validation

To compile an ADeL program, we first transform it into an equation system which represents the synchronous automaton as explained in 3.3. Then we implement directly this equation system, transforming it into a Boolean equation system thanks to the encoding defined in section 3.1 and to theorem 1. The latter system provides an effective implementation of the initial ADeL program.

Since the equations may not be independent, we need to find a valid order (compatible with their interdependencies) to be able to generate code for execution, simulation, and verification. Thus we defined an efficient sorting algorithm [7], using a critical path scheduling approach, which computes all the valid partial orders instead of one unique total order. This facilitates merging several equation systems, hence, we can perform a hierarchical compilation: we can include an already compiled and sorted code for a sub scenario into a main one, without recompiling the latter.

The internal representation as Boolean equation systems also makes it possible to verify and validate ADeL programs, by generating a format suitable for a dedicated model checker such as our own BLIF_CHECK[7]. The same internal representation also allows us to generate code for the off-the-shelf NuSMV model-checker[8].

4 Conclusion and Future work

In this report, we presented an efficient synchronous approach to describe (human) activities to be recognized by a computer system. The Synchronous Paradigm offers several advantages: generation of deterministic programs, concurrency management, well-established formal foundation, verification through model checking facilities, etc. We proposed a new simple user friendly synchronous language, ADeL, adapted to this model. We endowed this language with two equivalent formal semantics, one to describe

[7] `http://www.unice.fr/dgaffe/recherche/outils_blif.html`
[8] `http://nusmv.fbk.eu/`

the abstract behavior of a program, the second to compile the program into an automaton described as an equation system.

In the near future, our first objective is to simplify the use of ADeL for non computer scientists (e.g., physicians or security managers) We plan to provide more natural and easier to learn graphic interface. This is an ongoing project in collaboration with ergonomists.

Describing activities and generating the corresponding automata is just a part of a broader recognition system. This system aims at recognizing simultaneously several activities from various sensors, video analysis being one of the most important. The system is configured with the description of the different activities to recognized as produced by the ADeL compiler. At run time, it collects the objects and events detected by sensors; it determines the input environments of the automata; it dispatches these environments to the corresponding automata and triggers their reactions; finally, it reports the recognized activities based on automata output environments.

Our first tests show that the current code that ADeLs generates, basically composed of Boolean equations, is easy to integrate in the recognition system, produces compact code, and is efficient at run time. There remains a fundamental issue, common to all synchronous approaches: at the sensor level, the events are asynchronous and they must be sampled to constitute input environments and to define the synchronous "instants". No exact solution is available; we already tested several strategies and heuristics but large scale experiments are still necessary.

References

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

[2] 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.

[3] Daniel Gaffé and Annie Ressouche. Algebraic Framework for Synchronous Language Semantics. In Laviana Ferariu and Alina Patelli, editors, *2013 Symposium on Theoretical Aspects of Sofware Engineering*, pages 51–58, Birmingham, UK, July 2013. IEEE Computer Society.

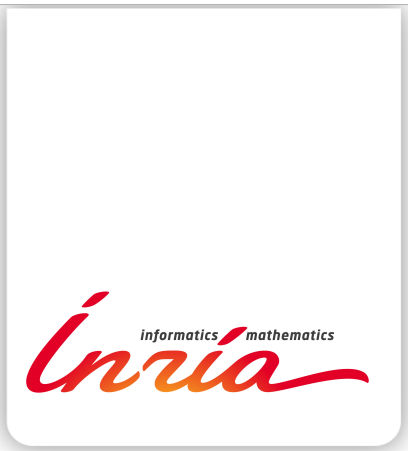
[4] Matthew Ginsberg. Multivalued logics: A uniform approach to inference in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

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

[6] G. H. Mealy. A method for synthesizing sequential circuits. *Bell Sys. Tech. Journal*, 34:1045–1080, September 1955.

[7] Annie Ressouche and Daniel Gaffé. Compilation modulaire d'un langage synchrone. *Revue des sciences et technologies de l'information, série Théorie et Science Informatique*, 4(30):441–471, June 2011.

[8] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399