Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format

Jan Stöcker, Frederic Lang, Hubert Garavel

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format

Jan Stöcker — Frédéric Lang — Hubert Garavel

N° 6950

Juin 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6950--FR+ENG

Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format

Jan Stöcker, Frédéric Lang, Hubert Garavel

Thème COM — Systèmes communicants
Projet Vasy

Rapport de recherche n° 6950 — Juin 2009 — 27 pages

Abstract: To model real-life critical systems, one needs "high-level" languages to express three important concepts: complex data structures, concurrency, and real-time. So far, the verification of timed systems has been successfully applied to "low-level" models, such as timed extensions of automata or of Petri nets. To bridge the gap between high-level languages, which allow a concise modeling of systems, and low-level models, for which efficient algorithms and tools have been designed, intermediate models are needed. In this report, we propose the ATLANTIF intermediate model, an extension with real-time and concurrency of the NTIF (*New Technology Intermediate Format*) intermediate model. We define the formal semantics of ATLANTIF and present a translator from ATLANTIF to timed automata (for verification using UPPAAL), and to time Petri nets (for verification using TINA).

Key-words: concurrency, formal method, intermediate model, real-time, time Petri net, timed automaton, verification

A short version of this report is available as "*Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format*", in M. Leuschel and H. Wehrheim (Eds.), Proceedings of the 7th International Conference on integrated Formal Methods iFM 2009 (Düsseldorf, Germany), February 16-19, 2009, LNCS 5423, Springer-Verlag.

Centre de recherche INRIA Grenoble – Rhône-Alpes
Inovallée, 655, avenue de l'Europe, Montbonnot, 38334 Saint Ismier Cedex (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Temps-réel et données dans des processus parallèles : Le format intermédiaire ATLANTIF

Résumé : Pour modéliser des systèmes critiques de manière réaliste, il est nécessaire de disposer de langages de "haut niveau", permettant d'exprimer trois concepts importants : des structures de données complexes, de la concurrence et du temps-réel. Jusqu'à présent, la vérification des systèmes temporisés a été appliquée avec succès à des modèles de "bas niveau", tels que des extensions temporelles des automates ou des réseaux de Petri. Pour combler le fossé entre des langages de haut niveau, qui permettent une modélisation concise de systèmes et des modèles de bas niveau, pour lesquels des outils et des algorithmes efficaces ont été développés, des modèles intermédiaires sont nécessaires. Dans ce rapport, nous proposons le modèle intermédiaire ATLANTIF, une extension avec temps-réel et concurrence du modèle intermédiaire NTIF (*New Technology Intermediate Format*). Nous définissons la sémantique formelle d'ATLANTIF et nous présentons un traducteur de ATLANTIF vers les automates temporisés (pour une vérification avec UPPAAL), et vers des réseaux de Petri temporisés (pour une vérification avec TINA).

Mots-clés : automate temporisé, concurrence, méthode formelle, modèle intermédiaire, temps-réel, réseau de Petri temporisé, vérification

1 Introduction

In many cases, asynchronous real-time systems can be modeled as a set of processes that run in parallel, communicate, synchronize mutually, and are subject to quantitative time constraints. The description and verification of asynchronous real-time systems has been a very active research subject, which has led to numerous theoretical results established upon various low-level models, such as timed automata [1, 11], timed extensions of Petri nets [32, 16], and timed process algebras [17, 31, 10, 18, 9, 40, 3, 34, 35, 28]. These models have been at the basis of successful verification tools, such as AltaRica [15], Kronos [41], Red [39], Romeo [26], Rtl [17], Tina [5], Uppaal [30], etc.

However, although appropriate for verification, these models are often too low-level for describing complex systems concisely. Higher-level models are thus needed. Such models should allow the expression of three aspects formally and simultaneously:

1. The first aspect is *data*, ranging from simple types (such as booleans, integers and enumerated types) to structured types (such a arrays, lists, unions, and trees). This also includes functions, either predefined or user defined.

2. The second aspect is *control*, such as communication, synchronization between processes, and the ability for processes to activate and/or deactivate each others.

3. The third aspect is *real-time*, such as *delays* (inaction of a process during a predefined time), constraints on the time instants when a process can communicate, *urgency* (indicating that a communication must not be delayed), and *latency* [17] (indicating that some time can elapse before a communication becomes urgent).

This scientific goal has been addressed since the late 80's, with the definition of high-level formal models that combine the strong theoretical foundations of process algebras with language features suitable for a wider industrial dissemination of formal methods [36, 31], converging into the E-Lotos language standardized by Iso [28]. On the other hand, several semi-formal industrial models based on model-driven tool development are emerging, such as Aadl [20], SysML [27] and Uml/Marte [19]. However, in both cases, verification tools are still lacking for these models. This could be adressed by translators from these high-level models to the low-level models accepted by existing verification tools. Suitable intermediate models are thus needed to enable a better integration of timed verification in industrial tool chains.

Related Work. Ntif (*New Technology Intermediate Form*) [22] is a minimal intermediate model for processes with sequential control and complex data. An Ntif process is an automaton that consists of a set of control states, to each of which is associated a statement called a *multibranch* transition and defined using high-level standard control structures (deterministic and nondeterministic variable assignments, **if-then-else** and **case** conditionals, nondeterministic choice, **while** loops, etc.) and communication events. This allows a representation of processes that is more compact than condition/action models such as If [14], Bip [4], and Lpes [37] and that can be easily translated into such models.

More recently, NTIF found industrial applications in the framework of the TOPCASED[1] project led by AIRBUS. The concepts of NTIF served as a basis for FIACRE (*Format Intermédiaire pour les Architectures de Composants Répartis Embarqués*) [6], an intermediate model between industrial models and verification tools. Transformations from AADL and SDL into FIACRE have been specified, and FIACRE has been connected to two model checkers: CADP [24] and TINA [8].

Contribution. As a basis to design the future revisions of FIACRE, we propose in this paper an enhanced version of NTIF named ATLANTIF, which provides more general concurrency and real-time constructs. As regards control, ATLANTIF provides a mechanism to synchronize, activate and deactivate processes, based on a generalization of synchronization vectors. As regards real-time, it associates delays and time constraints to communications, following the line of prior work that led to the definition of real-time process algebras, such as ET-LOTOS [31], RT-LOTOS [17], and E-LOTOS. ATLANTIF has a formal semantics that is intended to allow semantic-preserving translations from high-level languages into low-level models, and that satisfies suitable properties such as *time additivity* (every sequence of timed transitions can be collapsed into a single timed transition), *time determinism* (elapsing of a certain amount of time leads to a unique state) and *maximal progress of urgent actions* (time cannot elapse if an urgent action is possible) [33].

In order to assess our choices, we also present a prototype translator tool from ATLANTIF to lower-level models, thus enhancing the cooperation between different methods. It targets timed automata, suitable as input for the UPPAAL model checker and time Petri nets, suitable as input for the TINA model checker. We illustrate the benefits of ATLANTIF and its translators on four examples borrowed from the literature of real-time models.

Report outline. In Section 2, we present the syntax and formal semantics of ATLANTIF. In Section 3, we show how subsets of ATLANTIF can be translated into UPPAAL's timed automata and TINA's time Petri nets, we present a tool, and we give examples. In Section 4, we give some concluding remarks.

2 Overview of ATLANTIF

2.1 Syntax

The syntax of ATLANTIF, given in Fig. 1, is described in EBNF (*Extended Backus-Naur Form*), where parts between square brackets are optional and vertical bars denote alternatives. ATLANTIF is a strict superset of NTIF; shading is used to highlight these extensions, which will be detailed in Sections 2.2 and 2.3.

For conciseness, we will not detail type definitions (including complex data types, such as records, lists, etc.), type constructors, and function definitions.

[1] `http://www.topcased.org`

X ::= **module** M **is**
  [(**no** | **discrete** | **dense**) **time**] (*timing options*)
  type T_1 **is** $D_1 \ldots$ **type** T_n **is** D_n (*type declarations*)
  function F_1 **is** $Y_1 \ldots$ **function** F_k **is** Y_k (*function declarations*)
  $R_1 \ \ldots \ R_m$ (*synchronizers, defined below*)
  init $u_0, \ldots, u_j$ (*initially active units*)
  $U_0 \ \ldots \ U_l$ (*unit definitions, defined below*)
  end module

U ::= **unit** u **is**
  [**variables** $V_0 : T_0$ [:= E_0], ..., $V_n : T_n$ [:= E_n]] (*local variables*)
  from s_0 A_0 ... **from** s_m A_m (*list of transitions*)
  $U_1 \ \ldots \ U_l$ (*subunits*)
  end unit

A ::= $V_0, \ldots, V_n$ **:=** $E_0, \ldots, E_n$ (*deterministic assignment*)
  | $V_0, \ldots, V_n$ **:= any** $T_0, \ldots, T_n$ [**where** E] (*nondeterministic assignment*)
  | **reset** $V_0, \ldots, V_n$ (*variable reset*)
  | **wait** E (*delay*)
  | $G\ O_1 \ldots O_n$ [[**must** | **may**] **in** W] (*gate communication*)
  | **to** s' (*jump to state*)
  | A_1**;** A_2 (*sequential composition*)
  | **if** E **then** A_1 **else** A_2 **end** [**if**] (*conditional*)
  | **case** E **is** $P_0 \rightarrow A_0$ | ... | $P_n \rightarrow A_n$ **end** [**case**] (*deterministic choice*)
  | **select** A_0 `[]` ... `[]` A_n **end** [**select**] (*nondeterministic choice*)
  | **while** E **do** A_0 **end** [**while**] (*loop*)
  | **null** (*inaction*)

O ::= $!E$ (*value emission*)
  | $?P$ (*value reception*)

E ::= V (*variable*)
  | $F(E_1, \ldots, E_n)$ (*function*)
  | $C(E_1, \ldots, E_n)$ (*constructor*)

P ::= **any** T (*anonymous variable*) | P_0 **where** E (*condition*) | (P_0)
  | V (*variable*) | $C(P_1, \ldots, P_n)$ (*constructor*)

W ::= `[`E_1`,`E_2`]` | `]`E_1`,`E_2`]` | `[`E_1`,`E_2`[` | `]`E_1`,`E_2`[` (*bounded interval*)
  | `[`E_1`, ...[` | `]`E_1`, ...[` (*unbounded interval*)
  | W_1 **or** W_2 | W_1 **and** W_2 | `(`W_0`)` (*combined intervals*)

R ::= **sync** G [: B] **is** K (*synchronization formula*)
  [**stop** $u_1, \ldots, u_m$] [**start** $u'_1, \ldots, u'_n$] (*stopped and started units*)
  end sync

K ::= u (*single unit*)
  | K_1 **and** K_2 (*synchronization*)
  | K_1 **or** K_2 (*alternative*)
  | N **among** `(`$K_1, \ldots, K_m$`)`
  | `(`K_0`)`

N ::= n (*natural integer*)
  | N_1 **or** N_2 (*choice*)

B ::= **visible** | **hidden**
  | **urgent** | **silent**

where terminal and non terminal symbols mean the following:

A : *action*	M : *module identifier*	u : *unit identifier*
B : *visibility specifier*	N : *cardinality list*	U : *unit*
C : *constructor identifier*	O : *communication offer*	V : *variable identifier*
D : *type definition*	P : *pattern*	W : *time window*
E : *expression*	Q : *semantic modality*	X : *module (axiom)*
F : *function identifier*	R : *synchronizer*	Y : *function definition*
G : *gate identifier*	s : *state identifier*	
K : *synchronization formula*	T : *type identifier*	

Figure 1: ATLANTIF syntax (shading indicates additions w.r.t. NTIF)

2.2 Sequential processes in ATLANTIF

An ATLANTIF sequential process, called a *unit*, contains variable declarations and optionally *subunits*. We write $decl(u)$ the set of variables declared in a given unit u. Those can be read and/or written in the subunits of u, thus allowing variables to be shared between units. To this aim, each variable V has associated a scope, given by the set $accessible(V)$, that defines the units in which V can be read and/or written. Appendix A.1.3 defines this set formally.

Each unit contains a list of discrete states, the first of which is taken to be the initial state. To each discrete state s we associate a *multibranch* transition of the form "**from** s A", where A is an action, noted $act(s)$. Contrary to usual models, in which actions are simply "condition/assignment" pairs, ATLANTIF actions are built using high-level language constructs combining atomic actions. A particular action is *gate communication*, which allows data exchange in the form of offers, each of which represents either the emission ("$!E$") of some value expression E or the reception ("$?P$") of some value that is decomposed against a pattern P using pattern-matching.

As regards real-time, ATLANTIF supports either discrete time (corresponding to a time domain isomorphic to $\mathbb{N}$) or dense time (corresponding to $\mathbb{R}_{\geq 0}$), as well as untimed behaviour. This timing option is given in the header of a specification (by the keywords "**no time**", "**discrete time**", or "**dense time"**) and taken to be "**no time**" if unspecified. ATLANTIF also has a "**wait**" action allowing a given amount of time to elapse (borrowed from process algebras such as TCSP [36]), and the following optional additions to gate communication:

- A *time window* W that consists of intersections ("**and**") and unions ("**or**") of open or closed intervals, where "..." represents infinity. The communication may happen when the time elapsed since the communication action has been reached belongs to the time window. If W is unspecified, it is taken to be "$[0, ...[$". The time window thus has the role of a *life reducer*, similar to that found in different timed process algebras such as ET-LOTOS [31].

- A *modality* Q among "**must**" or "**may**", "**must**" indicating that the communication must occur before the end of the time window (which is called the *deadline*), and "**may**" indicating that time can elapse indefinitely. If unspecified, Q is taken to be "**may**". In the classification of [13], "**may**" corresponds to *weak* timed semantics, whereas "**must**" corresponds to *strong* timed semantics. Time Petri nets and FIACRE only allow strong timed semantics, whereas timed automata and most timed extensions of LOTOS allow a combination of both, which justifies our choice in ATLANTIF.

Static semantics. As regards static semantics, ATLANTIF inherits and extends the rules of NTIF [22]. The inherited rules concern well-typedness and restriction of at most one communication on each possible path of a multibranch transition. The extended rules concern the proper initialization of variables before use, which now has to take into account interaction and sharing of variables between units.

We add the constraints that no "**wait**" action is allowed in any path following a communication in a multibranch transition, that the time window of every

"**must**" communication is either unbounded or right-closed, that communications by **silent** synchronizers must not be restricted by time windows (i.e., they are understood to happen as early as they are possible), and that a unit cannot be active if one of its (direct or indirect) subunits is active. Furthermore, no unit that may write a variable V can be active at the same time as another unit that may read and/or write V. Appendix A gives formal definitions for the new and extended static semantics rules.

Dynamic semantics – definitions. As regards dynamic semantics, we need the following definitions inherited from NTIF. We assume a set *Val* of *values*, written v, v', v_0, v_1, etc. We note $\mathcal{V}$ the set of variables. Partial functions on $\mathcal{V} \rightarrow Val$, called *stores*, are written $\rho, \rho', \rho_0, \rho_1$, etc. We note $dom(\rho)$ the domain of ρ. The *update* operator $\oslash$ and the *restriction* operator $\ominus$ are defined on stores as follows:

$$\rho \oslash \rho' \stackrel{\text{def}}{=} \rho'' \text{ where } \rho''(V) = \text{if } V \in dom(\rho') \text{ then } \rho'(V) \text{ else } \rho(V)$$
$$\rho \ominus \{V_1, \ldots, V_n\} \stackrel{\text{def}}{=} \rho'' \text{ where } dom(\rho'') = dom(\rho) \setminus \{V_1, \ldots, V_n\} \text{ and } (\forall V \in dom(\rho''))\ \rho''(V) = \rho(V)$$

The semantics of expressions is given by a predicate $eval(E, \rho, v)$ that is **true** iff the evaluation of expression E in store ρ yields a value v. The semantics of patterns is given by a *pattern-matching* function $match(v, \rho, P)$ that returns either "**fail**" if v does not match P, or else a new store ρ' corresponding to ρ in which the variables of P have been assigned by the matching sub-terms of v. The semantics of offers is given by a function $accept(v, \rho, O)$, defined by:

$$accept(v, \rho, !E) \stackrel{\text{def}}{=} \text{if } eval(E, \rho, v) \text{ then } \rho \text{ else } \mathbf{fail}$$
$$accept(v, \rho, ?P) \stackrel{\text{def}}{=} match(v, \rho, P)$$

We note $\mathcal{S}$ the set of state identifiers assumed to contain a special element δ, reserved for semantics, which represents an auxiliary discrete state that denotes the termination of an action, thus enabling the execution of subsequent actions.

The following definitions are also required. We note $\mathbb{D}$ the time domain, t, t', t_0, t_1, etc. its elements, and $\mathbb{L}_1 \stackrel{\text{def}}{=} \{G\ v_1 \ldots v_n \mid G \in \mathbb{G},\ v_1, \ldots, v_n \in Val\} \cup \{\varepsilon\}$ the set of labels, where $\mathbb{G}$ denotes the set of gates and ε represents transitions without communication actions. The binary operator "+" is partially defined on $\mathbb{L}_1 \times \mathbb{L}_1 \rightarrow \mathbb{L}_1$ by $l + \varepsilon \stackrel{\text{def}}{=} l$, $\varepsilon + l \stackrel{\text{def}}{=} l$, and is undefined if both its operands are different from ε. We note $\mathbb{U}$ the set of unit identifiers and $\mathcal{U}$,$\mathcal{U}'$,$\mathcal{U}_0$,$\mathcal{U}_1$, etc. its subsets. We use the notation $\rho_{\restriction \mathcal{U}}$ to restrict the domain of store ρ to those variables that are accessible in a unit set $\mathcal{U}$, formally $\rho_{\restriction \mathcal{U}} \stackrel{\text{def}}{=} \rho \ominus \{V \mid (\forall u \in \mathcal{U})\ u \notin accessible(V)\}$. The semantics of time windows is given by a predicate $win\_eval(W, \rho, D)$ that is **true** iff the evaluation of W in store ρ yields a (possibly infinite) set of time instants D. We also define a boolean function $up\_lim(Q, W, \rho, t)$ returning **true** iff $Q =$ **must** and the set D defined by $win\_eval(W, \rho, D)$ has a maximum equal to t.

Dynamic semantics – sequential constructs. In NTIF, the semantics of actions was defined by a relation of the form $(A, \rho) \stackrel{l}{\Longrightarrow} (s, \rho')$, where A is an action, ρ, ρ' are stores, $s \in \mathcal{S}$ is a discrete state, and $l \in \mathbb{L}_1$ is a label [22].

ATLANTIF extends this to a relation of the form $(A, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho')$, where d, d' have the form (t, μ), with t a time value (intuitively representing the time that may elapse in the current unit until the next communication), and μ a boolean (called *blocking condition*), that is equal to **true** iff time is not allowed to elapse after t. This means that the action A in the context d and ρ evolves to the *local state* (s, d', ρ') (local states are also written $\sigma, \sigma', \sigma_0, \sigma_1$, etc.), producing a transition labeled l. These rules are detailed below, where shading indicates additions w.r.t. NTIF.

$$(null)\ \frac{}{(\mathbf{null}, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, d, \rho)} \qquad (wait)\ \frac{eval(E, \rho, v) \wedge t \geq v}{(\mathbf{wait}\ E, (t, \mu), \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, (t - v, \mu), \rho)}$$

$$(assign_d)\ \frac{eval(E_0, \rho, v_0) \wedge \ldots \wedge eval(E_n, \rho, v_n)}{(V_0, \ldots, V_n := E_0, \ldots, E_n, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, d, \rho \oslash [V_0 \mapsto v_0, \ldots, V_n \mapsto v_n])}$$

$$(assign_n)\ \frac{v_0 \in T_0, \ldots, v_n \in T_n \wedge \rho' = \rho \oslash [V_0 \mapsto v_0, \ldots, V_n \mapsto v_n] \wedge eval(E, \rho', \mathbf{true})}{(V_0, \ldots, V_n := \mathbf{any}\ T_0, \ldots, T_n\ \mathbf{where}\ E, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, d, \rho')}$$

$$(reset)\ \frac{}{(\mathbf{reset}\ V_0, \ldots, V_n, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, d, \rho \ominus \{V_0, \ldots, V_n\})} \qquad (to)\ \frac{}{(\mathbf{to}\ s, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (s, d, \rho)}$$

$$(comm)\ \frac{(\forall j \in 1..n)\ accept(v_j, \rho_j, O_j) = \rho_{j+1} \neq \mathbf{fail} \wedge win\_eval(W, \rho_{n+1}, D) \wedge t \in D}{(G\ O_1 \ldots O_n\ Q\ \mathbf{in}\ W, (t, \mu), \rho_1) \stackrel{G\ v_1 \ldots v_n}{\Longrightarrow} (\delta, (t, up\_lim(Q, W, \rho_{n+1}, t)), \rho_{n+1})}$$

$$(seq_1)\ \frac{(A_1, d, \rho) \stackrel{l_1}{\Longrightarrow} (\delta, d', \rho') \wedge (A_2, d', \rho') \stackrel{l_2}{\Longrightarrow} \sigma}{(A_1; A_2, d, \rho) \stackrel{l_1 + l_2}{\Longrightarrow} \sigma} \qquad (seq_2)\ \frac{(A_1, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho') \wedge s \neq \delta}{(A_1; A_2, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho')}$$

$$(select)\ \frac{k \in 0..n \wedge (A_k, d, \rho) \stackrel{l}{\Longrightarrow} \sigma}{(\mathbf{select}\ A_0\ [] \ldots []\ A_n\ \mathbf{end}, d, \rho) \stackrel{l}{\Longrightarrow} \sigma}$$

$$(case)\ \frac{\begin{array}{c} eval(E, \rho, v)\ \wedge\ (\forall j < k)\ match(v, \rho, P_j) = \mathbf{fail} \\ \wedge\ match(v, \rho, P_k) = \rho_k \wedge\ (A_k, d, \rho_k) \stackrel{l}{\Longrightarrow} \sigma \end{array}}{(\mathbf{case}\ E\ \mathbf{is}\ P_0 \rightarrow A_0 \mid \ldots \mid P_n \rightarrow A_n\ \mathbf{end}, d, \rho) \stackrel{l}{\Longrightarrow} \sigma}$$

$$(while_1)\ \frac{eval(E, \rho, \mathbf{true})\ \wedge\ (A; \mathbf{while}\ E\ \mathbf{do}\ A\ \mathbf{end}, d, \rho) \stackrel{l}{\Longrightarrow} \sigma}{(\mathbf{while}\ E\ \mathbf{do}\ A\ \mathbf{end}, d, \rho) \stackrel{l}{\Longrightarrow} \sigma}$$

$$(while_2)\ \frac{eval(E, \rho, \mathbf{false})}{(\mathbf{while}\ E\ \mathbf{do}\ A\ \mathbf{end}, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, d, \rho)}$$

$$(\varepsilon\text{-}elim)\ \frac{(A, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (s, d', \rho') \wedge s \neq \delta \wedge (act(s), d', \rho') \stackrel{l}{\Longrightarrow} (s', d'', \rho'')}{(A, d, \rho) \stackrel{l}{\Longrightarrow} (s', d'', \rho'')}$$

Fig. 2 gives an example of a system composed of a user and a lamp. The user, modeled by the *User* unit, pushes repeatedly a button using gate *Push*. Between two pushes, the user may wait indefinitely, but must wait at least one time unit. The lamp, modeled by the *Lamp* unit, has three levels of brightness, modeled by the three discrete states *Off*, *Low*, and *Bright*. When the lamp is off (state *Off*), pushing the button switches it on with low brightness (state *Low*). If the next push happens within less than 5 time units then the lamp gets brighter (state *Bright*). If it happens after 5 time units then the lamp is switched off.

```
module Light is dense time
sync Push is User and Lamp end sync
init User, Lamp (* initially started units *)
unit User is
  from Rdy
    wait 1; Push; to Rdy
end unit
unit Lamp is
  from Off
    Push; to Low
  from Low
    select Push in [0, 5[;
           to Bright
        [] Push in [5, ... [;
           to Off
    end select
  from Bright
    Push; to Off
end unit
end module
```

Figure 2: ATLANTIF program describing a light switch

2.3 Concurrency in ATLANTIF

In ATLANTIF, a specification contains several units synchronized with respect to *synchronizers* (Fig. 1), which are a generalization of synchronization vectors [2, 12]. A synchronizer is invoked every time a unit reaches a communication action i.e., every time it wants to propose a rendezvous to its environment. It describes how units synchronize and determines the set of running units, which are called *active*. Precisely, a synchronizer has the form "**sync** $G : B$ **is** K **stop** $u_1, \ldots, u_m$ **start** $u'_1, \ldots, u'_n$ **end sync**", where:

- G is a gate that triggers the synchronizer.
- B is an optional tag attached to G, noted $tag(G)$, which may take one out of four different values: "**visible**" induces a transition labeled by G and the offers exchanged on G; "**hidden**" induces an internal transition called τ-transition; "**urgent**" behaves like the latter, but also blocks time when a synchronization is possible; and "**silent**" indicates that the communication does not induce a transition. If no tag is specified, the synchronizer is visible.
- K is a formula consisting of unit identifiers and boolean operators, which denotes combinations of units that must synchronize, each such combination being called a "*synchronization set*". The set of synchronization sets attached to G, noted $sync(G)$, is defined as follows:

$$
\begin{array}{l}
sync(u) = \{\{u\}\} \\
sync(K_1 \textbf{ and } K_2) = \{S_1 \cup S_2 \mid S_1 \in sync(K_1) \wedge S_2 \in sync(K_2)\} \\
sync(K_1 \textbf{ or } K_2) = sync(K_1) \cup sync(K_2) \\
sync(n \textbf{ among } (K_1, \ldots, K_m)) = sync(K'_1 \textbf{ or} \ldots \textbf{or } K'_k), \text{ where} \\
\quad \{K'_1, \ldots, K'_k\} = \{(K_{i_1} \textbf{ and} \ldots \textbf{and } K_{i_n}) \mid 1 \leq i_1 < \ldots < i_n \leq m\} \\
sync(n_1 \textbf{ or} \ldots \textbf{or } n_l \textbf{ among } (K_1, \ldots, K_m)) = \\
\quad sync(n_1 \textbf{ among } (K_1, \ldots, K_m) \textbf{ or} \ldots \textbf{or } n_l \textbf{ among } (K_1, \ldots, K_m))
\end{array}
$$

- "**stop** $u_1, \ldots, u_m$" and "**start** $u'_1, \ldots, u'_n$" are optional constructs indicating that the units $u_1, \ldots, u_m$ become inactive, whereas $u'_1, \ldots, u'_n$ become active when the synchronizer is triggered. We note $stop(G) = \{u_1, \ldots, u_m\}$ and $start(G) = \{u'_1, \ldots, u'_n\}$. By default, $stop(G) = \varnothing$ and $start(G) = \varnothing$.

To express concurrency, other intermediate models (such as CÆSAR networks [21] or communicating state machines [29]) combine communications of

processes into Petri net-like transitions. A drawback of this approach is that the number of transitions in the resulting model can be the product of the numbers of transitions in each process. Synchronizers provide a more symbolic approach that avoids these problems, while being general enough to express the following:

- Competition between synchronizing processes can be expressed by synchronizers denoting several synchronization sets e.g., in the formula "u_1 **and** (u_2 **or** u_3)", u_2 and u_3 compete to synchronize with u_1.
- *Multiway* synchronization can be expressed by synchronization sets containing more than two units e.g., in "u_1 **and** u_2 **and** u_3", the three units u_1, u_2 and u_3 must synchronize altogether.
- The generalized parallel composition operators of [25] can also be expressed. For instance, "**par** $G\#2, G\#3$ **in** $u_1||u_2||u_3$ **end par**", which means that either two or three processes among u_1, u_2, and u_3 synchronize on G, can be expressed by "**sync** G **is** 2 **or** 3 **among** (u_1, u_2, u_3) **end sync**".
- Processes can be started and/or stopped by themselves or by concurrent processes. For instance, "**sync** G **is** u_1 **and** u_2 **stop** u_1, u_2 **start** u_3, u_4 **end sync**" means that units u_1, u_2 are stopped as soon as they synchronize on gate G, and that u_3, u_4 are started at the same moment.

Dynamic semantics – concurrency and real-time. Contrary to NTIF, which had no parallel semantics as it was limited to sequential processes, ATLANTIF supports a second layer of semantics for concurrency and real-time. It is given by a TLTS (*Timed Labeled Transition System*) of the form $(\mathbb{S}, \mathbb{T}, S_0)$, where:

- $\mathbb{S}$ is a set of *global states* (as opposed to the *local states*) of the form (π, θ, ρ) (written S, S', S_0, S_1, etc.), where $\pi : \mathbb{U} \to \mathcal{S}$ is a partial function, called *state distribution*, that maps each active unit to its current discrete state, $\theta : \mathbb{U} \to (\mathbb{D} \times \textbf{Bool})$ is a partial function, called *time distribution*, that maps each active unit to its current time value and blocking condition, and ρ is a store. Note that the set of active units is given by $dom(\pi)$ and $dom(\theta)$, with $dom(\pi) = dom(\theta)$.
- $\mathbb{T}$ is a set of transitions defined as a relation in $\mathbb{S} \times \mathbb{L}_2 \times \mathbb{S}$, where $\mathbb{L}_2 \stackrel{\text{def}}{=} (\mathbb{L}_1 \setminus \{\varepsilon\}) \cup \{\tau\} \cup (\mathbb{D} \setminus \{0\})$. Transitions labeled in $\mathbb{D} \setminus \{0\}$ are called *timed* transitions, whereas the other transitions are called *discrete* transitions.
- $S_0 \in \mathbb{S}$ is the initial state, which is defined by $S_0 \stackrel{\text{def}}{=} (\pi_{0|\mathcal{U}_0}, \theta_{0|\mathcal{U}_0}, \rho_{0|\mathcal{U}_0})$, where π_0 is a function that maps each unit to its initial discrete state (defined implicitly as the first discrete state in the corresponding unit), θ_0 is the function that constantly returns $(0, \textbf{false})$ for each unit, ρ_0 is the store that maps each variable to its initial value, if any, and $\mathcal{U}_0$ is the set of initially active units (see Fig. 1). $\pi_{0|\mathcal{U}_0}$ and $\theta_{0|\mathcal{U}_0}$ represent respectively π_0 and θ_0 whose domain is restricted to $\mathcal{U}_0$.

A discrete transition corresponds to a chain of zero or more silent synchronizations, followed by a non-silent synchronization (hereafter called a *chain*).

We also call *incomplete chain* the prefix of a chain. A chain executes without time elapsing.

As observed in [21], it would be incorrect to explore all possible chains. Instead, a silent synchronization is allowed in a chain only if at least one of the synchronizing and not stopped units or one of the started units are also synchronizing in another synchronization later in the chain. We call the *unit affection* of a synchronization the set consisting of the synchronizing and not stopped units and of the started units. We associate to each incomplete chain the set α of unit affections corresponding to synchronizations in the incomplete chain, no unit of which has synchronized later in the incomplete chain. Only those chains ending with an empty set of unit affections must be explored.
We define the following predicates:

- The predicate $synchronizing((S,\alpha),l,\mu,(S',\alpha'))$, defined on $(\mathbb{S}\times 2^{2^{\mathbb{U}}})\times(\mathbb{L}_1\setminus\{\varepsilon\})\times\mathbf{Bool}\times(\mathbb{S}\times 2^{2^{\mathbb{U}}})$, is **true** iff (1) a transition labeled l may occur in global state S and leads to global state S', (2) the disjunction of the blocking conditions in the local states reached via this transition equals μ, and (3) a set of unit affections α evolves to α' via this synchronization. Formally:

$$
\begin{array}{l}
synchronizing(((\pi,\theta,\rho),\alpha),G\ v_1\ldots v_n,\mu,((\pi',\theta',\rho'),\alpha')) \stackrel{\text{def}}{=}\\
\quad(\exists\{u_1,\ldots,u_m\}\in sync(G))\ \{u_1,\ldots,u_m\}\subseteq dom(\pi)\ \wedge\\
\qquad(\forall i\in 1..m)((act(\pi(u_i)),\theta(u_i),\rho_{\restriction\{u_i\}})\stackrel{G\ v_{1..m}}{\Longrightarrow}(s_i,(t_i,\mu_i),\rho_i)\wedge s_i\neq\delta)\wedge\\
\qquad\mu=\bigvee_{i=1..m}\mu_i\ \wedge\\
\qquad next\_\pi(\pi,[u_i\mapsto s_i\mid i\in 1..m],G,\pi')\ \wedge\\
\qquad next\_\theta(\theta,\{u_1,\ldots,u_m\},min_{i\in 1..m}(t_i),G,\theta')\ \wedge\\
\qquad next\_\rho(\rho,[V\mapsto\rho_i(V)\mid u_i\in accessible(V)],dom(\pi'),G,\rho')\ \wedge\\
\qquad next\_\alpha(\alpha,\{u_1,\ldots,u_m\},G,\alpha')
\end{array}
$$

The predicate $next\_\pi$ defines the new state distribution after a synchronization.

$$
\begin{array}{l}
next\_\pi(\pi,\pi_1,G,\pi')\stackrel{\text{def}}{=}\\
\qquad\pi'=((\pi\oslash\pi_1)\ominus stop(G))\oslash[u\mapsto\pi_0(u)\mid u\in start(G)]
\end{array}
$$

The predicate $next\_\theta$ defines the new time distribution after a synchronization. If the synchronization was **silent**, the new time value t_0 of the affected units is given by the minimal time value of the synchronizing units. This corresponds to the unit(s), in which the longest delay took place i.e., the unit(s) for which the other synchronizing units had to wait. Otherwise, the new time value of the affected units is set to 0.

$$
\begin{array}{l}
next\_\theta(\theta,\mathcal{U},t_0,G,\theta')\stackrel{\text{def}}{=}\\
\qquad\theta'=\begin{cases}((\theta\oslash[u\mapsto(t_0,\mathbf{false})\mid u\in\mathcal{U}]))\ominus stop(G)\\
\qquad\oslash[u\mapsto(t_0,\mathbf{false})\mid u\in start(G)] & \text{if } tag(G)=\mathbf{silent}\\
((\theta\oslash[u\mapsto(0,\mathbf{false})\mid u\in\mathcal{U}])\ominus stop(G))\\
\qquad\oslash[u\mapsto(0,\mathbf{false})\mid u\in start(G)] & \text{otherwise}\end{cases}
\end{array}
$$

The predicate $next\_\rho$ defines the new store after a synchronization. The domain of the store is restricted according to the new set of active units. For each started unit declaring a variable V with initial value (see Fig. 1), the new store is extended by $[V\mapsto\rho_0(V)]$.

$$next\_\rho(\rho, \rho_1, \mathcal{U}, G, \rho') \stackrel{\text{def}}{=} \rho' = ((\rho \oslash \rho_1) \ominus \{V \mid (\forall u \in \mathcal{U})\ u \notin accessible(V)\}) \oslash [V \mapsto \rho_0(V) \mid V \in dom(\rho_0) \wedge (\exists u \in start(G))\ V \in decl(u)]$$

The predicate $next\_\alpha$, which defines the new unit affections after a synchronization, begins by deleting those unit affections from which at least one unit has synchronized. If the synchronization was **silent**, then a new set is added, containing all synchronizing and not stopped units and all started units.

$$next\_\alpha(\alpha, \mathcal{U}, G, \alpha') \stackrel{\text{def}}{=} \alpha' = \begin{cases} (\alpha \setminus \{\mathcal{U}' \in \alpha \mid (\exists u \in \mathcal{U})\ u \in \mathcal{U}'\}) \cup \\ \qquad \{(\mathcal{U} \setminus stop(G)) \cup start(G)\} & \text{if } tag(G) = \textbf{silent} \\ \alpha \setminus \{\mathcal{U}' \in \alpha \mid (\exists u \in \mathcal{U})\ u \in \mathcal{U}'\} & \text{otherwise} \end{cases}$$

where the operators $\oslash$, $\ominus$ are defined on π and θ in a similar way as on ρ.

- The predicate $enabled(S, l, \mu, S')$, defined on $\mathbb{S} \times (\mathbb{L}_1 \setminus \{\varepsilon\}) \times \textbf{Bool} \times \mathbb{S}$, is **true** iff there is a chain that has to be explored, starting in global state S and ending in global state S', where the last synchronization is labeled l and the blocking condition reached via this synchronization equals μ. Formally:

$$\begin{aligned} &enabled(S, l, \mu, S') \stackrel{\text{def}}{=} (\exists S_1, \ldots, S_k, \alpha_1, \ldots, \alpha_k, l_1, \ldots, l_k, \mu_1, \ldots, \mu_k) \\ &\quad synchronizing((S, \varnothing), l_1, \mu_1, (S_1, \alpha_1)) \wedge \ldots \wedge \\ &\qquad synchronizing((S_k, \alpha_k), l_k, \mu_k, (S', \varnothing)) \wedge \\ &\quad tag(l_1) = \ldots = tag(l_{k-1}) = \textbf{silent} \wedge tag(l_k) = l \neq \textbf{silent} \wedge \mu_k = \mu \end{aligned}$$

- Time cannot elapse in a global state if an urgent communication is enabled i.e., a chain terminates with a communication on a gate whose synchronizer is tagged urgent or a chain terminates with a communication of the form "$G\ O_1 \ldots O_n$ **must in** W" when the deadline of W has been reached. The predicate $relaxed(S)$, defined on $\mathbb{S}$, is **true** iff time can elapse in S. Formally:

$$\begin{aligned} &relaxed(S) \stackrel{\text{def}}{=} (\forall\ G\ v_1 \ldots v_n, \mu, S') \\ &\qquad enabled(S, G\ v_1 \ldots v_n, \mu, S') \Rightarrow (\neg\mu \wedge tag(G) \neq \textbf{urgent}) \end{aligned}$$

Discrete transitions are defined by rule (rdv) as follows:

$$(rdv)\ \frac{enabled((\pi, \theta, \rho), G\ v_1 \ldots v_n, \mu, (\pi', \theta', \rho'))}{(\pi, \theta, \rho) \xrightarrow{label(G\ v_1 \ldots v_n)} (\pi', \theta', \rho')}$$

where function $label$ transforms a non-ε label of $\mathbb{L}_1$ into a discrete label of $\mathbb{L}_2$:

$$label(G\ v_1 \ldots v_n) \stackrel{\text{def}}{=} \text{ if } tag(G) = \textbf{visible} \text{ then } G\ v_1 \ldots v_n \text{ else } \tau$$

Timed transitions are defined by rule $(time)$, which allows t units of time to elapse as long as no urgent communication is enabled. The new state is calculated by increasing all relative times by t, using "+" defined by $(\forall u)\ (\theta + t)(u) \stackrel{\text{def}}{=} (t_u + t, \mu_u)$ where $\theta(u) = (t_u, \mu_u)$.

$$(time)\ \frac{t > 0 \wedge (\forall\ t' < t)\ relaxed((\pi, \theta + t', \rho))}{(\pi, \theta, \rho) \xrightarrow{t} (\pi, \theta + t, \rho)}$$

We illustrate the semantics by deriving two TLTS transitions for the light switch example shown in Fig. 2, page 9. We show that when *User* is in state *Rdy* and *Lamp* in state *Low*, 3 time units may elapse before the button is pushed. Formally: $(\pi, \theta, \varnothing) \xrightarrow{3} (\pi, \theta + 3, \varnothing) \xrightarrow{Push} (\pi \oslash [Lamp \mapsto Bright], \theta, \varnothing)$, where $\pi \stackrel{\text{def}}{=} [User \mapsto Rdy, Lamp \mapsto Low]$, and $\theta \stackrel{\text{def}}{=} [User \mapsto (0, \mathbf{f}), Lamp \mapsto (0, \mathbf{f})]$ (where **f** is a shorthand for **false**).

First, $(\pi, \theta, \varnothing) \xrightarrow{3} (\pi, \theta + 3, \varnothing)$ comes from the following derivation:

$$\frac{3 > 0 \wedge (\forall t' < 3) relaxed((\pi, \theta + t', \varnothing))}{(\pi, \theta, \varnothing) \xrightarrow{3} (\pi, \theta + 3, \varnothing)}(time)$$

Second, $(\pi, \theta + 3, \varnothing) \xrightarrow{Push} (\pi \oslash [Lamp \mapsto Bright], \theta, \varnothing)$ comes from:

$$\frac{\{User, Lamp\} \in sync(Push) \wedge (act(Rdy), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing) \wedge (act(Low), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Bright, (3, \mathbf{f}), \varnothing)}{(\pi, \theta + 3, \varnothing) \xrightarrow{Push} (\pi \oslash [Lamp \mapsto Bright], \theta, \varnothing)}(rdv)$$

The premiss $(act(Rdy), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)$ comes from the following, recalling that $act(Rdy) =$ "**wait** 1; *Push*; **to** *Rdy*":

$$\frac{\dfrac{eval(1, \varnothing, 1) \wedge 3 \geq 1}{(\textbf{wait } 1, (3, \mathbf{f}), \varnothing) \stackrel{\varepsilon}{\Longrightarrow} (\delta, (2, \mathbf{f}), \varnothing)}(wait) \quad (Push;\ \textbf{to } Rdy, (2, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)}{(act(Rdy), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)}(seq_1)$$

At last, the premiss $(Push;\ \textbf{to } Rdy, (2, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)$ comes from:

$$\frac{\dfrac{}{(Push, (2, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (\delta, (2, \mathbf{f}), \varnothing)}(comm) \quad \dfrac{}{(\textbf{to } Rdy, (2, \mathbf{f}), \varnothing) \stackrel{\varepsilon}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)}(to)}{(Push;\ \textbf{to } Rdy, (2, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)}(seq_1)$$

The premiss $(act(Low), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Bright, (3, \mathbf{f}), \varnothing)$ is derived similarly by the rules (*comm*), (*to*), (seq_1), and (*select*).

With this semantic approach, we respect the standard property that time must elapse at the same speed in all units. Furthermore, the following proposition shows that this semantics has the suitable properties mentioned in Section 1.

Proposition. The TLTS corresponding to the semantics of an ATLANTIF specification satisfies the properties of (i) time additivity (two successive delays are equal to their sum), (ii) time determinism (no state allows two different successors after the same delay) and (iii) maximal progress of urgent actions (no delay is possible in states where an urgent action is possible).

Proof. (i) Let S, S' be global states. We must show that $\forall\ t_1, t_2 \in (\mathbb{D} \setminus \{0\})$:

$$S \xrightarrow{t_1 + t_2} S' \text{ iff } (\exists\ S'')\ S \xrightarrow{t_1} S'' \text{ and } S'' \xrightarrow{t_2} S'$$

We define $S \stackrel{\text{def}}{=} (\pi, \theta, \rho)$. We note that time can only elapse using the (*time*) rule, which does not modify π and ρ and increases θ by some delay. Therefore, the above statement can be rephrased as:

$$(\pi, \theta, \rho) \xrightarrow{t_1 + t_2} (\pi, \theta + (t_1 + t_2), \rho)$$
$$\text{iff} \quad (\pi, \theta, \rho) \xrightarrow{t_1} (\pi, \theta + t_1, \rho) \text{ and } (\pi, \theta + t_1, \rho) \xrightarrow{t_2} (\pi, (\theta + t_1) + t_2, \rho)$$

Given the definition of $+$, it is obvious that $\theta + (t_1 + t_2) = (\theta + t_1) + t_2$. From the premiss of rule (*time*), we can reduce the above goal to the obvious following statement:

$$(\forall\ t' < t_1 + t_2)\ relaxed((\pi, \theta + t', \rho))$$
$$\text{iff } (\forall\ t' < t_1)\ relaxed((\pi, \theta + t', \rho)) \text{ and } (\forall\ t' < t_2)\ relaxed((\pi, \theta + (t_1 + t'), \rho))$$

(ii) Again, we note that time can only elapse using rule (*time*), which for given global state S and time t defines a unique successor state.

(iii) Let S be a global state allowing an urgent action, i.e. $\neg relaxed(S)$. Then the premiss of rule (*time*) cannot be satisfied in S i.e., time cannot elapse in S. □

3 Automated Translations to Verification Tools

We developed a prototype translator tool, which maps ATLANTIF models to either the TA (*timed automata*) used by the tool UPPAAL [30] or the TPN (*time Petri nets*) used by TINA [8]. Outlines of these mappings are given in this section. We assume the reader is familiar with UPPAAL's TA and TINA's TPN.

Common restrictions. Some concepts of ATLANTIF cannot be mapped to neither UPPAAL's TA nor TINA's TPN. Concretely, ATLANTIF models must use dense time; expressions in **wait** actions and time windows must be integer constants; nondeterministic assignments are not supported; patterns must be made up of either variables or constants exclusively. In addition, **while** loops are not yet supported in the translation to TA, although the translation would be feasible. The elimination of **silent** synchronizers does not exist in UPPAAL or TINA; we translate them with unlabeled transitions instead, which obviously changes the semantics.

Translation to UPPAAL. Each ATLANTIF unit is mapped to a TA. Each discrete state s is mapped to a TA location (also named s) and an invariant is synthesized from the **must** constraints of multibranch transitions originating from s. The action $act(s)$ is decomposed into one TA transition for each branch of control. If a gate communication admits several synchronization sets containing the current unit, then it is split into one transition for each such synchronization set. Since TA do not allow communication offers, data exchanges are emulated using TA shared variables.

A key issue is that UPPAAL's TA synchronizations involve at most two automata[2], whereas ATLANTIF allows multiway synchronizations involving $n > 2$ units. The solution requires that exactly one unit sends data (i.e., all offers are emissions), whereas the $(n-1)$ other units receive data (i.e., all offers are receptions): the gate communication in the sender unit is split into a sequence of $(n-1)$ communications, each of which synchronizes with a receiver.

We also have to emulate the starting and stopping of units. To this aim, every TA corresponding to a unit that is stopped or started by at least one synchronizer receives one auxiliary location named "disabled". The sequence of communications in an "emission unit" described above is followed by one communication labeled by a broadcast channel "*G*_stop !" if the synchronizer stops units, and by one communication labeled by a broadcast channel "*G*_start !" if the related synchronizer starts units. Each unit that is stopped by G receives an additional transition to location "disabled" labeled "*G*_stop ?" in each location. Each unit that is started by G receives one additional transition labeled "*G*_start ?" from "disabled" to the initial location.

[2] UPPAAL also allows a broadcast communication, which is inapt for our purpose, because UPPAAL's broadcast is not blocking.

Translation to TINA. Each ATLANTIF unit is mapped to a TPN. Each discrete state s is mapped to a TPN place (also named s) and the corresponding action $act(s)$ is decomposed into several TPN transitions, each TPN transition being labeled by a gate. As regards time constraints, we only consider time intervals and we implement a solution inspired from [7], that requires additional auxiliary places and transitions. Given a communication on a gate G, which corresponds to a Petri net transition T, we calculate the sum m of all delays that occur in "**wait**" actions preceding the communication. We remove these wait actions and we increase the bounds of the time window by m. The resulting time window is then implemented in the form of zero, one, or two new transitions as follows:

- If the lower bound of the time window is $n > 0$, then we add an unlabeled transition with time constraint "$[n, \omega[$" (or "$]n, \omega[$", if the bound is strict), no out-place and a new in-place s_1. We add s_1 both to the inhibitor places of T, and to the out-places of every transition for which s is already an out-place.
- If the modality of the communication is **may** and the time window has an upper bound n, then we add an unlabeled transition with time constraint "$]n, \omega[$" (or "$[n, \omega[$", if the bound is strict), no out-place and a new in-place s_2. We add s_2 to the in-places of T and the new transition is given priority over T.
- If the modality of the communication is **must** and the time window has an upper bound n, then we add an unlabeled transition with time constraint "$[n, n]$", no out-place and a new in-place s_3. We add s_3 to the in-places of T and all transitions except those created for other **must** constraints are given priority over this new unlabeled transition.

The TPNs corresponding to each unit are combined into a single one by merging synchronizing transitions, using the method described in [7]. If such a transition is labeled by a synchronizer that stops some units, then the out-places belonging to these units are deleted. If it is labeled by a synchronizer that starts some units, then the initial places of these units become new out-places. This encoding requires that the stopped units belong to the synchronization set.

Tool implementation. Our prototype translator was implemented using the method proposed in [23] and consists of 538 lines of C code, 2,193 lines of SYNTAX code, and 13,146 lines of LOTOS NT code. The tool architecture is schematized in Fig. 3.

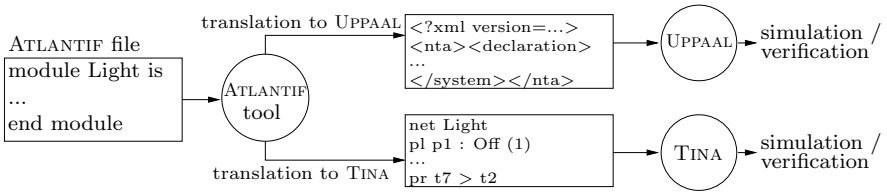

Figure 3: The ATLANTIF to UPPAAL / TINA translation tool

We applied this translator to four examples, namely the light switch presented in Fig. 2 (page 9), the CSMA/CD protocol, which is a common benchmark specification [41], a stop-and-wait protocol, implemented with one sender, one receiver and two transmission channels, and a train gate controller. The translations into TA and TPN of the light switch example are shown in Fig. 4 and 5 respectively.

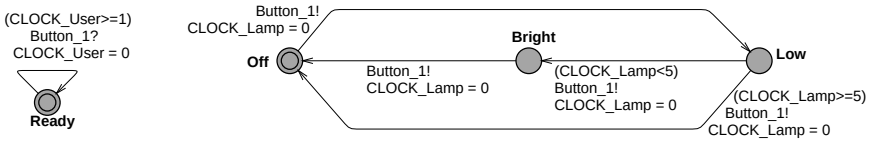

Figure 4: The two automatically generated UPPAAL TA for the light switch example

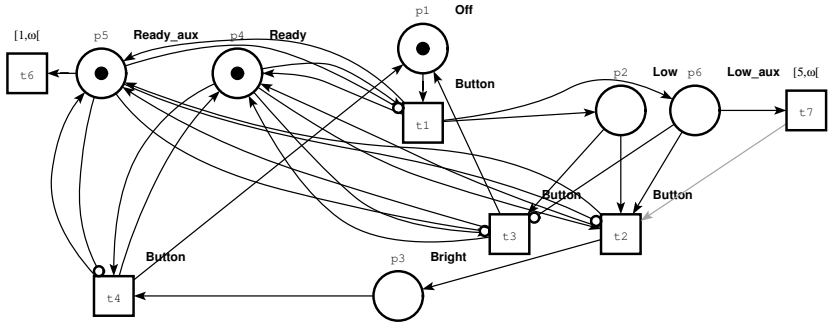

Figure 5: The automatically generated TINA time Petri net for the light switch example

Fig. 6 compares the size of ATLANTIF programs with the size of the corresponding TA and TPN. It shows that ATLANTIF enables shorter descriptions, in particular due to its concise syntax for time and its ability to define multiway synchronizations. Note that the number of locations of the TA generated for the CSMA/CD is the same as in a handwritten specification available on the web[3].

	ATLANTIF		UPPAAL-TA		TINA-TPN	
	disc. states	trans.	locations	trans.	places	trans.
Light switch	4	4	4	5	6	6
CSMA/CD (3 Stations)	12	12	14	42	40	142
Stop-and-wait	10	10	10	12	29	56
Train Gate Controller	12	12	18	18	23	18

Figure 6: Size comparison: ATLANTIF vs. generated UPPAAL vs. generated TINA

These results suggest that the TA translation is efficient for programs with multiple occurrences of simple synchronizers (i.e., synchronizers involving at

[3] http://www.it.uu.se/research/group/darts/uppaal/benchmarks/#CSMA

most two units), whereas the TPN translation is efficient for limited occurrences of more complex synchronizers.

4 Conclusion

This report proposes ATLANTIF, a simple and elegant extension of the intermediate model NTIF [22] with concurrency and real-time, intended for a better integration of formal verification tools in industrial environments. Thus, ATLANTIF supports the three main concepts needed to model complex asynchronous real-time systems: elaborate data types, concurrency, and quantitative time.

ATLANTIF has a simple timed semantics, where time elapsing is concentrated in a single rule, which satisfies time additivity, time determinism, and maximal progress. This goal is not obvious to achieve: for example, complex syntactic restrictions had to be brought to E-LOTOS to ensure those properties; as another example, RT-LOTOS does not satisfy time additivity.

We also presented a translator mapping ATLANTIF to two advanced verification tools, UPPAAL [30] and TINA [8].

As regards future work, we plan to extend our translator with new features and to use it on larger industrial examples. ATLANTIF could also be a basis to enhance the FIACRE intermediate model [6] used in the TOPCASED project.

References

[1] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[2] André Arnold. MEC: A System for Constructing and Analysing Transition Systems. In Joseph Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 117–132. Springer Verlag, June 1989.

[3] J. Baeten and C. Middelburg. *Process Algebra with Timing: Real Time and Discrete Time*. In *Handbook of Process Algebra*, chapter 10, pages 627–648. North-Holland, 2001.

[4] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12. IEEE Computer Society, 2006.

[5] B. Berthomieu and M. Diaz. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[6] B. Berthomieu, H. Garavel, F. Lang, and F. Vernadat. Verifying Dynamic Properties of Industrial Critical Systems Using TOPCASED/FIACRE. *ERCIM News*, 75:32–33, October 2008.

[7] B. Berthomieu, F. Peres, and F. Vernadat. Bridging the gap between Timed Automata and Bounded Time Petri Nets. In *FORMATS*, LNCS 4202. Springer-Verlag, 2006.

[8] B. Berthomieu and F. Vernadat. Time Petri Nets Analysis with TINA. In *3rd International Conference on The Quantitative Evaluation of Systems (QEST)*, 2006.

[9] Stefan Blom, Natalia Ioustinova, and Natalia Sidorova. Timed verification with μCRL. In Manfred Broy and Alexandre V. Zamulin, editors, *Proceedings of the 5th Andrei Ershov International Conference on Perspectives of System Informatics PSI'2003 (Novosibirsk, Russia)*, volume 2890 of *Lecture Notes in Computer Science*, pages 178–192. Springer Verlag, July 2003. Also available as CWI Research Report SEN-E0312, Amsterdam, December 2003.

[10] T. Bolognesi and F. Lucidi. LOTOS-like Process Algebras with Urgent or Timed Interactions. In Kenneth R. Parker and Gordon A. Rose, editors, *Proceedings of the 4th International Conference on Formal Description Techniques FORTE'91*. North-Holland, 1991.

[11] S. Bornot, J. Sifakis, and S. Tripakis. Modelling Urgency in Timed Systems. In *COMPOS*, LNCS, 1997.

[12] Amar Bouali, Annie Ressouche, Valérie Roy, and Robert de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*. Springer Verlag, August 1996.

[13] M. Boyer and O. H. Roux. Comparison of the Expressiveness of Arc, Place and Transition Time Petri Nets. In *Petri Nets and Other Models of Concurrency – ICATPN 2007*, volume LNCS 4546, pages 63–82. Springer-Verlag, 2007.

[14] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. Tools and Applications II: The IF Toolset. In *SFM*, 2004.

[15] F. Cassez, C. Pagetti, and O. Roux. A timed extension for AltaRica. *Fundamenta Informaticæ*, 62(3-4):291–332, August 2004.

[16] A. Cerone and A. Maggiolo-Schettini. Time-based expressivity of Time Petri Nets for system specification. *Theoretical Computer Science*, 216(1):1–54, 1999.

[17] J.-P. Courtiat and R. Cruz de Oliveira. On RT-LOTOS and its Application to the Formal Design of Multimedia Protocols. *Annals of Telecommunications*, 50(11-12):888–906, Nov/Dec 1995.

[18] J. W. Davies and S. A. Schneider. A Brief History of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, February 1995.

[19] M. Faugère, T. Bourbeau, R. de Simone, and S. Gérard. MARTE: Also an UML Profile for Modeling AADL Applications. In *ICECCS*. IEEE, 7 2007.

[20] P. Feiler, D. Gluch, and J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical note, Carnegie Mellon, 2 2006.

[21] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[22] Hubert Garavel and Frédéric Lang. NTIF: A General Symbolic Model for Communicating Sequential Processes with Data. In Doron Peled and Moshe Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2002 (Houston, Texas, USA)*, volume 2529 of *Lecture Notes in Computer Science*, pages 276–291. Springer Verlag, November 2002. Full version available as INRIA Research Report RR-4666.

[23] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compiler Construction using LOTOS NT. In Nigel Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction CC 2002 (Grenoble, France)*, volume 2304 of *Lecture Notes in Computer Science*, pages 9–13. Springer Verlag, April 2002.

[24] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer Verlag, July 2007.

[25] Hubert Garavel and Mihaela Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In Jianping Wu, Qiang Gao, and Samuel T. Chanson, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'99 (Beijing, China)*, pages 185–202. IFIP, Kluwer Academic Publishers, October 1999.

[26] G. Gardey, D. Lime, M. Magnin, and O. Roux. Roméo: A tool for analyzing time Petri nets. In *17th International Conference on Computer Aided Verification (CAV'05)*, Lecture Notes in Computer Science, July 2005.

[27] M. Hause. The SysML Modelling Language. In *Fifteenth European Systems Engineering Conference*, 9 2006.

[28] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.

[29] Günter Karjoth. Implementing LOTOS Specifications by Communicating State Machines. In *Proceedings of the third International Conference on Concurrency Theory (CONCUR'92)*, volume 630 of *Lecture Notes in Computer Science*, pages 386–400. Springer Verlag, August 1992.

[30] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1 - 2):134–152, October 1997.

[31] L. Léonard and G. Leduc. A Formal Definition of Time in LOTOS. *Formal Aspects of Computing*, pages 28–96, 1998.

[32] P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, University of California, Irvine, Dep. of Information and Computer Science, 1974.

[33] Xavier Nicollin and Joseph Sifakis. An Overview and Synthesis on Timed Process Algebras. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[34] Xavier Nicollin and Joseph Sifakis. The Algebra of Timed Processes ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994.

[35] J. Ouaknine and J. Worrell. Timed CSP = closed timed ε-automata. *Nordic Journal of Computing*, 10(2):99–133, 2003.

[36] G. M. Reed and A. W. Roscoe. A Timed Model for Communicating Sequential Processes. *Theoretical Computer Science*, 58:249–261, 1988.

[37] Michel A. Reniers and Yaroslav S. Usenko. Analysis of Timed Processes with Data Using Algebraic Transformations. In Jan Chomicki and David Toman, editors, *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning TIME'05 (Burlington, Vermont, USA)*. IEEE, 2005.

[38] J. Stöcker. *An Intermediate Model for the Verification of Asynchronous Real-Time Embedded Systems: Definition and Application of the* ATLANTIF *language.* PhD thesis, Grenoble INP, 2009. To appear.

[39] F. Wang. Symbolic Simulation Checking of Dense-Time Automata. In *5th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)*, LNCS, Salzburg, Austria, October 2007. Springer-Verlag.

[40] Wang Yi. CCS + Time = An Interleaving Model for Real Time Systems. In *Automata, Languages and Programming, 18th International Colloquium*, volume 510 of *LNCS*, pages 217–228, 1991.

[41] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, October 1997.

A Static semantics

In this appendix, we give the formal definitions of some of the static semantics rules of ATLANTIF. They complement the formal definitions given in the appendix of [22].

A.1 Unit hierarchy and variables

A.1.1 Active units

In this section, we give the static semantics rules assuring that there can never be two active units in a global state such that one is a (direct or indirect) subunit of the other. We call *well-activated* an ATLANTIF module that satisfies this property, which is an important condition to avoid variable access conflicts between units (see Section A.1.3).

Since a unit can be declared as a subunit of another unit, we can define a partial order "$\succ$" on $\mathbb{U}$, by $u \succ u'$ iff u' is a direct or indirect subunit of u. We write $u \succeq u'$ for $u \succ u'$ or $u = u'$. We use $\succ$ to formally define the following predicate *valid_active* on $2^{\mathbb{U}}$, which expresses that no unit in a set is a subunit of another unit in the same set:

$$\mathit{valid\_active}(\mathcal{U}) \stackrel{\text{def}}{=} (\forall u, u' \in \mathcal{U})\ u \not\succ u'$$

We say that a global state (π, θ, ρ) is *well-activated* iff $\mathit{valid\_active}(\mathit{dom}(\pi))$.

When a synchronization on a gate G leads from a well-activated state S to a state S', it has to be assured that S' is also well-activated. To this aim, it is sufficient to demand the following:

- A unit that is active in S cannot be started by G, unless it is also stopped by G.
- The union of the units started by G with the units active in S without the units stopped by G must be *valid_active*.

Formally, this is given by the following predicate, where the set $\mathcal{U}$ corresponds to the units active in S:

$$\begin{aligned}&\mathit{validity\_stable}(G) \stackrel{\text{def}}{=}\\ &\quad(\forall \mathcal{U} \subseteq \mathbb{U})\ (\mathit{valid\_active}(\mathcal{U}) \wedge (\exists \mathcal{U}' \subseteq \mathcal{U})\ \mathcal{U}' \in \mathit{sync}(G)\\ &\qquad\Rightarrow (\mathcal{U} \setminus \mathit{stop}(R)) \cap \mathit{start}(G) = \varnothing \wedge \mathit{valid\_active}((\mathcal{U} \setminus \mathit{stop}(G)) \cup \mathit{start}(G)))\end{aligned}$$

To define a synchronizer that satisfies this predicate, the list of stopped units should therefore contain each unit that can be in conflict with one of the started units. This predicate is defined by a quantification on the power set of $\mathbb{U}$, thus a naïve implementation would have an exponential complexity. The tool implementation of ATLANTIF uses therefore an alternative but equivalent predicate, which induces an algorithm of polynomial complexity. Moreover, the implemented algorithm automatically detects and lists all units that have to be stopped additionally, in order to establish validity-stability.

An ATLANTIF module is *well-activated*, if its set of initially active units $\mathcal{U}_0$ satisfies $\mathit{valid\_active}(\mathcal{U}_0)$ and if each synchronizer G satisfies $\mathit{validity\_stable}(G)$.

A.1.2 Binding

Appendix A.1 of [22] formally defines *well-binding* i.e., every variable occurence can be bound unambiguously to its declaration. This includes the definition of the following sets:

- The sets $use(E)$, $use(P)$, and $use(O)$ contain the variables *used* (i.e., read) in an expression, a pattern, and an offer respectively. For ATLANTIF, we extend this definition to an interval W:

$$use(W) \stackrel{\text{def}}{=} \begin{cases} use(E_1) \cup use(E_2) & \text{if } W \text{ has the form } \texttt{[}E_1\texttt{,}E_2\texttt{]}, \\ & \quad \texttt{[}E_1\texttt{,}E_2\texttt{[}, \texttt{]}E_1\texttt{,}E_2\texttt{]}, \text{ or } \texttt{]}E_1\texttt{,}E_2\texttt{[} \\ use(E) & \text{if } W \text{ has the form } \texttt{[}E\texttt{,}\ldots\texttt{[} \\ & \quad \text{or } \texttt{]}E\texttt{,}\ldots\texttt{[} \\ use(W_1) \cup use(W_2) & \text{if } W \text{ has the form } (W_1 \textbf{ and } W_2) \\ & \quad \text{or } (W_1 \textbf{ or } W_2) \end{cases}$$

- The sets $def(E)$, $def(P)$, and $def(O)$ contain the variables *defined* (i.e., written) in an expression, a pattern, and an offer respectively. For ATLANTIF, we extend this definition to an interval W:

$$def(W) \stackrel{\text{def}}{=} \varnothing$$

We say that a variable V is *used* in an action A, if A contains an expression E, a pattern P, an offer O, or a time window W such that $V \in use(E)$, $V \in use(P)$, $V \in use(O)$, or $V \in use(W)$ respectively. V is *used* in a unit u, if u contains one discrete state s such that V is used in $act(s)$. We write $use(u)$ the set of variables used in u.

A variable V is *defined* in an action A, if A contains a pattern P or an offer O such that $V \in def(P)$ or $V \in def(O)$ respectively. V is *defined* in a unit u, if u contains one discrete state s such that V is used in $act(s)$ or if V is declared in u and assigned with an initial value (see Fig. 1). We write $def(u)$ the set of variables defined in u.

Furthermore, we extend the definition of action *well-binding* as follows:

- Every action of the form "**wait** E" is well-bound.
- Every action of the form "$G\ O_1 \ldots O_n\ Q$ **in** W" is well-bound if:
  - for all O_i of the form "?P_i", P_i is well-bound, and
  - for all $i < j$, $def(O_i) \cap def(O_j) = \varnothing$ and $def(O_i) \cap use(O_j) = \varnothing$, and
  - $\bigcup_{i \in 1..n} def(O_i) \cap use(W) = \varnothing$.

We say that a unit u is well-bound iff for each variable V in $use(u) \cup def(u)$, there exists a $u' \succeq u$ such that $V \in decl(u')$. To avoid name clashes, we thus assume that all variables have distinct names.

Finally, an ATLANTIF module is well-bound iff each of its units is well-bound.

A.1.3 Variable access conflicts

ATLANTIF offers the possibility to share variables between different units, which in general can lead to ambiguous behaviour caused by variable access conflicts.

For instance, let u_1 be a unit declaring a variable V and let u_2, u_3 be subunits of u_1. If u_2 contains a multibranch transition "**from** s_1 $V := 5; G;$ **to** s_2" and u_3 contains a multibranch transition "**from** s_3 $V := 3; G;$ **to** s_4", then a synchronization on G by these transitions would assign to V the values 5 and 3 at the same time.

We forbid such a situation by defining in this section the notion of *well-accessible* modules.

A variable V, declared in a unit u_0, defined in the units $u_1, \ldots, u_n$, and used in the units $u'_1, \ldots, u'_m$ is *well-accessible*, if it satisfies the following two constraints:

- To avoid write/write conflicts, in every global state at most one of the units $u_1, \ldots, u_n$ may be active simultaneously. Supposing the module is well-activated, it is sufficient to demand that the set $\{u_1, \ldots, u_n\}$ is totally ordered by $\succ$.
- To avoid read/write conflicts, if a unit u'_i with $1 \leq i \leq m$ and $u'_i \notin \{u_0, u_1, \ldots, u_n\}$ is active in a global state, no unit of $\{u_0, u_1, \ldots, u_n\}$ may be active in the same global state. Supposing the module is well-activated, it is sufficient to demand that $\{u'_i, u_0, u_1, \ldots, u_n\}$ is totally ordered by $\succ$.

If the variable V is well-accessible, we can define the set $\mathit{accessible}(V) \subseteq \mathbb{U}$ of all units in which V may be read and/or written. Formally:

$\mathit{accessible}(V) \stackrel{\text{def}}{=} \{u \in \mathbb{U} \mid u_0 \succeq u \wedge \{u, u_1, \ldots, u_n\}$ is totally ordered by $\succ\}$

An ATLANTIF module is well-accessible if each of its variables is well-accessible.

A.1.4 Variable initialization

Appendix A.3 of [22] formally defines an algorithm that checks whether variables in a sequential NTIF process are systematically defined before they are used. This algorithm cannot be applied to the units of an ATLANTIF module, because variables can be shared between units e.g., defined in one unit and used in another. Thus, we need a new algorithm that applies to the whole module.

The algorithm we present in this section can be seen as an extension of the abovementioned one and consists of three steps.

First step: We begin by constructing for each unit u one directed graph, called local *variable usage graph*. Each discrete state s in u maps to a node (also named s) in the graph. The multibranch transition associated to s maps to zero or more edges, the set of which is formally defined by $\mathit{state\_local\_edges}(s) \stackrel{\text{def}}{=} \{(\mathcal{V}_o, \mathcal{V}_a, \mathcal{V}_r, l, s') \in \mathit{local\_edges}(\mathit{act}(s)) \mid s' \neq \delta\}$ where function *local_edges* is formally defined in Fig. 7. Each $(\mathcal{V}_o, \mathcal{V}_a, \mathcal{V}_r, l, s') \in \mathit{state\_local\_edges}(s)$ represents one possible execution path of $\mathit{act}(s)$ that terminates with a jump to s'. It corresponds to an edge from s to s', with label $l \in (\mathbb{G} \cup \{\varepsilon\})$, and three associated sets of variables as follows:

- The set $\mathcal{V}_o$ contains those variables that need to have a value assigned before this path is taken.

- The set $\mathcal{V}_a$ contains those variables that are necessarily defined at the end of this path.
- The set $\mathcal{V}_r$ contains those variables that are potentially reset at the end of this path.

$$
\begin{array}{l}
\mathit{local\_edges}(\textbf{null}) \stackrel{\text{def}}{=} \{(\varnothing,\varnothing,\varnothing,\varepsilon,\delta)\} \\
\mathit{local\_edges}(\textbf{wait } E) \stackrel{\text{def}}{=} \{(\mathit{use}(E),\varnothing,\varnothing,\varepsilon,\delta)\} \\
\mathit{local\_edges}(V_0,\ldots,V_n \textbf{:=} E_0,\ldots,E_n) \stackrel{\text{def}}{=} \{(\bigcup_{i\in 0..n}\mathit{use}(E_i),\{V_0,\ldots,V_n\},\varnothing,\varepsilon,\delta)\} \\
\mathit{local\_edges}(V_0,\ldots,V_n \textbf{:=any } T_0,\ldots,T_n \textbf{ where } E) \stackrel{\text{def}}{=} \\
\qquad\qquad \{(\mathit{use}(E),\{V_0,\ldots,V_n\},\varnothing,\varepsilon,\delta)\} \\
\mathit{local\_edges}(\textbf{reset } V_0,\ldots,V_n) \stackrel{\text{def}}{=} \{(\varnothing,\varnothing,\{V_0,\ldots,V_n\},\varepsilon,\delta)\} \\
\mathit{local\_edges}(G\ O_1\ldots O_n\ Q\ W) \stackrel{\text{def}}{=} \\
\qquad\qquad \{(\bigcup_{i\in 0..n}\mathit{use}(O_i)\cup\mathit{use}(W),\bigcup_{i\in 0..n}\mathit{def}(O_i),\varnothing,G,\delta)\} \\
\mathit{local\_edges}(\textbf{to } s') \stackrel{\text{def}}{=} \{(\varnothing,\varnothing,\varnothing,\varepsilon,s')\} \\
\mathit{local\_edges}(\textbf{select } A_0 \,[]\, \ldots \,[]\, A_n \textbf{ end}) \stackrel{\text{def}}{=} \bigcup_{i\in 0..n}\mathit{local\_edges}(A_i) \\
\mathit{local\_edges}(\textbf{case } E \textbf{ is } P_0\text{->}A_0 \mid \ldots \mid P_n\text{->}A_n \textbf{ end}) \stackrel{\text{def}}{=} \\
\quad \bigcup_{i\in 0..n}\{(\mathcal{V}_o\cup\mathit{use}(E)\cup\mathit{use}(P_i),\mathcal{V}_a\cup\mathit{def}(P_i),\mathcal{V}_r,l,s') \mid \\
\qquad\qquad (\mathcal{V}_o,\mathcal{V}_a,\mathcal{V}_r,l,s')\in\mathit{local\_edges}(A_i)\} \\
\mathit{local\_edges}(\textbf{if } E \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ end}) \stackrel{\text{def}}{=} \\
\quad \{(\mathcal{V}_o\cup\mathit{use}(E),\mathcal{V}_a,\mathcal{V}_r,l,s') \mid (\mathcal{V}_o,\mathcal{V}_a,\mathcal{V}_r,l,s')\in\mathit{local\_edges}(A_1)\cup\mathit{local\_edges}(A_2)\} \\
\mathit{local\_edges}(\textbf{while } E \textbf{ do } A_0 \textbf{ end}) \stackrel{\text{def}}{=} \{(\mathit{use}(E)\cup\bigcup_{i\in 0..n}\mathcal{V}_o^i,\varnothing,\bigcup_{i\in 0..n}\mathcal{V}_r^i,\varepsilon,\delta)\} \\
\qquad \text{where } \mathit{local\_edges}(A_0)=\{(\mathcal{V}_o^i,\mathcal{V}_a^i,\mathcal{V}_r^i,\varepsilon,\delta)\mid i\in 0..n\} \\
\mathit{local\_edges}(A_1;A_2) \stackrel{\text{def}}{=} \{(\mathcal{V}_o\cup(\mathcal{V}_o'\setminus\mathcal{V}_a),(\mathcal{V}_a\setminus\mathcal{V}_r')\cup\mathcal{V}_a',(\mathcal{V}_r\setminus\mathcal{V}_a')\cup\mathcal{V}_r',l_1+l_2,s_2) \mid \\
\qquad (\mathcal{V}_o,\mathcal{V}_a,\mathcal{V}_r,l_1,\delta)\in\mathit{local\_edges}(A_1),(\mathcal{V}_o',\mathcal{V}_a',\mathcal{V}_r',l_2,s_2)\in\mathit{local\_edges}(A_2)\}\cup \\
\quad \{(\mathcal{V}_o,\mathcal{V}_a,\mathcal{V}_r,l_1,s_1)\in\mathit{local\_edges}(A_1)\mid s_1\neq\delta\} \\
\qquad \text{raise error, if } \mathcal{V}_r\cap\mathcal{V}_o'\neq\varnothing
\end{array}
$$

Figure 7: Function *local_edges*

Second step: The local variable usage graphs of the units are then composed into a single (global) variable usage graph, where each node represents a state distribution function and each edge represents either a multibranch transition path in a single unit without communication action taken, or one or several paths in one unit each with communication action G such that the set of these units is in $\mathit{sync}(G)$. During the latter case, units may be stopped and/or started, according to G.

The algorithm given in Fig. 8 shows the construction of the global variable usage graph, where the latter is written as a set *VUG*. For each node π, *VUG* contains one tuple $(\pi,\mathcal{K})$, where $\mathcal{K}$ is the set of edges from node π. Each $(\mathcal{V}_o,\mathcal{V}_a,\mathcal{V}_r,\pi')\in\mathcal{K}$ is an edge from π to π' and with the three variable sets as above. The algorithm starts in node (π_0), defined as in Section 2.3 on page 10. It uses the predicate *next_π* from Section 2.3.

Third step: Given a variable usage graph *VUG* with initial node π_0 and the corresponding tuple $n_0=(\pi_0,\mathcal{K}_0)$, a greatest fix-point algorithm, formally defined in Fig. 9, calculates for each of its tuples $n=(\pi,\mathcal{K})$ a set of variables *set_before*(n). This set corresponds to the smallest possible set of variables that

$VUG \leftarrow \varnothing$
$Tmp \leftarrow \{(\pi_0)\}$
while ($Tmp \neq \varnothing$) do
  $new\_edges \leftarrow \varnothing$
  choose (π) from Tmp
  $Tmp \leftarrow Tmp \setminus \{(\pi)\}$
  $to\_sync \leftarrow \varnothing$
  for each $u \in dom(\pi)$
    $\mathcal{E} \leftarrow state\_local\_edges(\pi(u))$
    $\mathcal{E}_u \leftarrow \varnothing$
    for each $(\mathcal{V}_o, \mathcal{V}_a, \mathcal{V}_r, l, s) \in \mathcal{E}$
      if $l = \varepsilon$ then
        $new\_edges \leftarrow new\_edges \cup \{(\mathcal{V}_o, \mathcal{V}_a, \mathcal{V}_r, (\pi \oslash [u \mapsto s]))\}$
        if $((\pi \oslash [u \mapsto s])$ not in $VUG)$ then $Tmp \leftarrow Tmp \cup \{(\pi \oslash [u \mapsto s])\}$
      else
        $\mathcal{E}_u \leftarrow \mathcal{E}_u \cup \{(\mathcal{V}_o, \mathcal{V}_a, \mathcal{V}_r, l, s)\}$
      end if
    end for
    $to\_sync \leftarrow to\_sync \cup \{\mathcal{E}_u\}$
  end for
  for each synchronizer G
    for each $\mathcal{U}' \in sync(G)$
      if $(\mathcal{U}' = \{u_1, \ldots, u_n\} \subseteq dom(\pi))$ then
        $lost\_by\_disabling \leftarrow$
            $\{V \mid (\nexists u \in ((dom(\pi) \setminus stop(G)) \cup start(G)))\ u \in accessible(V)\}$
        $init\_by\_enabling \leftarrow \{V \mid V \in dom(\rho_0) \wedge (\exists u \in start(G))\ V \in decl(u)\}$
        for each choice of $(\mathcal{V}_o^1, \mathcal{V}_a^1, \mathcal{V}_r^1, G, s_1) \in \mathcal{E}_{u_1}, \ldots,$
            $(\mathcal{V}_o^n, \mathcal{V}_a^n, \mathcal{V}_r^n, G, s_n) \in \mathcal{E}_{u_n}$ with $\mathcal{E}_{u_1}, \ldots, \mathcal{E}_{u_n} \in to\_sync$
          $next\_node \leftarrow (\pi')$ where $next\_\pi(\pi, [u_i \mapsto s_i \mid i \in 1..n], G, \pi')\}$
          $new\_edges \leftarrow new\_edges \cup \{(\bigcup_{i \in 1..n} \mathcal{V}_o^i, \bigcup_{i \in 1..n} \mathcal{V}_a^i \cup init\_by\_enabling,$
            $\bigcup_{i \in 1..n} \mathcal{V}_r^i \cup lost\_by\_disabling, next\_node)\}$
          if $(next\_node$ not in $VUG)$ then $Tmp \leftarrow Tmp \cup \{next\_node\}$
        end for
      end if
    end for
  end for
  $VUG \leftarrow VUG \cup \{(\pi, new\_edges)\}$
end while

Figure 8: Pseudocode for variable usage graph construction

are defined in a global state with the state distribution π. At the same time, the algorithm checks for all outgoing edges (i.e., edges in $\mathcal{K}$), if the variables that need to be defined for these edges are always defined in π i.e., if the set $\mathcal{V}_o$ of each edge is a subset of *set_before*(n). If not, this means that there is possibly a path where an undefined variable is used.

```
for each n ∈ VUG
   set_before(n) ← V
   explored(n) ← false
end for
set_before(n0) ← ∅
while ((∃n ∈ VUG) explored(n) = false)
   choose n ∈ VUG with explored(n) = false
      explored(n) = true
      for each (Vo, Va, Vr, π') ∈ K  (where n = (π, K))
         if ((∃V ∈ Vo) V ∉ set_before(n) then raise error end if
         find n' = (π', K') ∈ VUG  (for the π' given above)
         Tmp ← set_before(n') ∩ ((set_before(n) \ Vr) ∪ Va)
         if ((Tmp ≠ set_before(n')) or (explored(n') = false)) then
            set_before(n') ← Tmp
            explored(n') ← false
         end if
      end for
   end choose
end while
```

Figure 9: Pseudocode for fix-points of variable definitions in variable usage graph

A.2 Other static semantics rules

Unicity of communication and next state reachability. The rule that each execution path of a multibranch transition contains at most one communication action and each communication action is necessarily followed by a jump (appendix A.4 of [22]) remains unchanged w.r.t. NTIF.

Typing. Appendix A.2 of [22] defines *well-typed* actions. We extend this definition as follows: The expression E in an action of the form "**wait** E" as well as the expressions E_1 and E_2 occuring in an interval `[`E_1`,`E_2`]`, `[`E_1`,`E_2`[`, `]`E_1`,`E_2`]`, `]`E_1`,`E_2`[`, `[`E_1`,...]`, or `]`E_1`,...]` of a communication action must either have type **natural** (if the timing option of the module is "**discrete time**") or type **float** (if the timing option of the module is "**dense time**").

No delay after a discrete transition. No **wait** action is allowed in any multibranch transition path following a communication action. This means that each communication action is followed instantly by a jump to another discrete state. Thus, this restriction assures that the global state after a discrete transition is well defined.

Restriction of timing constraints to discrete transitions. Time windows control the time elapsing between two discrete (i.e., **visible**, **hidden**, or

urgent) synchronizations. Communication actions using a **silent** synchronizer induce semantically invisible events, thus they must not have a time window.

The dynamic semantics rules suppose silent communications to happen as soon as they become possible. Note that it is allowed to space them using **wait** actions.

Time windows and strong timed semantics. The following constraints must be satisfied:

- All intervals in the time window of a "**must**" communication action must have one of the forms `[`E_1`,`E_2`]`, `]`E_1`,`E_2`]`, `[`E_1`,...[`, or `]`E_1`,...[`. In other words, intervals of the form `[`E_1`,`E_2`[` or `]`E_1`,`E_2`[` are forbidden in time windows.

  This restriction guarantees that whenever a synchronization on a **must** communication is possible, the time that may still elapse can be determined. Otherwise, for instance, in the action "G **must in** `[0,3[`" (supposing a dense time module) it would not be possible to determine the maximal amount of time that may elapse, since the interval $[0, 3[$ has no maximum.

- All intervals in the time window of a communication action using an **urgent** synchronizer must have one of the forms `[`E_1`,`E_2`]`, `[`E_1`,`E_2`[`, or `[`E_1`,...[`. In other words, intervals of the form `]`E_1`,`E_2`]`, `]`E_1`,`E_2`[`, or `]`E_1`,...[` are forbidden in time windows.

  This restriction excludes many constructs that might lead to deadlock states i.e., states that allow neither synchronizations nor time elapsing. Otherwise, for instance, the action "G **in** `]0,...[`" in a dense time module with G **urgent** would induce a deadlock, because some time would have to elapse before a communication on G being enabled, but time elapsing is not permitted by the semantics. A more detailed discussion on deadlock problems in ATLANTIF can be found in [38].

Centre de recherche INRIA Grenoble – Rhône-Alpes
Inovallée, 655, avenue de l'Europe, Montbonnot - 38334 Saint Ismier Cedex (France)

Centre de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes, 4, rue Jacques Monod - Bât. G - 91893 Orsay Cedex (France)
Centre de recherche INRIA Nancy – Grand Est : 615, rue du Jardin Botanique - 54600 Villers-lès-Nancy (France)
Centre de recherche INRIA Rennes – Bretagne Atlantique : Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis (France)

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