Institut Polytechnique
de Grenoble

Collège doctoral

École doctorale MSTII

An Intermediate Model for the Verification of Asynchronous Real-Time Embedded Systems:

Definition and Application of the ATLANTIF language

THÈSE

présentée et soutenue publiquement le 10 décembre 2009

pour l'obtention du

Doctorat de Grenoble INP

(spécialité informatique)

par

Jan STÖCKER

Composition du jury

Président :	Roland GROZ
Rapporteurs :	Elie NAJM François VERNADAT
Examinateur :	Alain GRIFFAULT
Co-Directeur de thèse :	Frédéric LANG
Co-Directeur de thèse :	Hubert GARAVEL

Institut National de Recherche en Informatique et en Automatique

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche GRENOBLE - RHÔNE-ALPES

Remerciements

Tout d'abord, je remercie mon directeur de thèse Frédéric Lang pour m'avoir proposé le sujet de thèse et de m'y avoir guidé à travers les discussions très intéressantes que nous avons eues. Je tiens à le remercier aussi pour sa patience (ce qui n'était toujours pas facile avec moi) et sa bonne humeur. Je remercie aussi Hubert Garavel pour m'avoir acceulli dans son équipe-projet VASY de l'INRIA et pour m'avoir donné beaucoup d'idées importantes.

Les collaborateurs de l'équipe VASY – Olivier, Anton, Marie et tous les autres ont rendu mon temps à l'INRIA agréable et plein de bons souvenirs. L'aide apportée par les assistantes de l'équipe, et surtout par Helen, concernant tous les grands et petits soucis administratifs, a eu une valeur immense pour moi.

Merci beaucoup aux membres de mon jury, d'avoir accepté ce travail, merci surtout aux rapporteurs pour leurs remarques constructives et leurs idées pour l'amélioration de ce manuscrit.

Beaucoup de mes amis, dont plusieurs que j'ai eu la chance de rencontrer pendant mes trois années passées à Grenoble, ont été à mes côtés dans des temps difficiles et dans des temps agréables. Loin de ma propre famille, mes colocataires ont été une deuxième famille pour moi.

Chaleureusement je dis merci – Herzlichen Dank ! – à mes parents et aux autres membres de ma famille, pour m'avoir toujours encouragé, pour les visites mutuelles et pour beaucoup de petites et grandes choses.

Pour ma mère,
parce qu'elle m'a fait découvrir les mathématiques.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 General context

In the year 2009, the world is more dependent than ever on computer systems, and this dependency is likely to grow in the future. Money that only exists electronically, the virtual elimination of distances by cellphones and the Internet, and many other characteristics of the fundamental changes in our culture are nowadays accepted as a normal part of life.

Technically, much of this has become possible by broad usage of *embedded systems* i.e., very specialised computer systems integrated in electric or electronic devices and controlling these devices. The tasks they perform are usually too complex to be carried out by human beings (e.g., fast calculations on large amounts of input data), or too dangerous (e.g., on-board control of space vessels, manipulations in nuclear reactors or chemical factories), or simply too tedious (e.g., control of traffic lights). Often, many embedded systems are connected and communicate with each other.

Obviously, these numerous dependencies induce risks and vulnerabilities, as failures of critical systems can lead to serious effects (e.g., unnecessary traffic jams) or even disastrous consequences (e.g., aircraft crashes). To avoid this, a developer cannot rely exclusively on common sense, because the parallel execution of several embedded systems easily reaches a complexity that goes beyond the scope of human imagination.

Therefore, it is essential to find a systematic means to ensure that hard- and software systems work correctly. When a failure during the usage cannot be risked, it is clear that these means have to be placed in the design process of such a system. Among the approaches that exist to this end is to apply a formal method such as *model checking*.

Model checking begins by describing a system's behaviour, which includes aspects such as what messages sends to other systems or to human users, what messages it receives, how the input data is proceeded, and how much time elapses before a next step is taken.

Writing this description is done using a standardized notation, which may either be purely textual, or a mixture of textual and graphical elements, and which is defined with unambiguous, formal semantics.

The next step is to state the properties that must be satisfied in the system e.g., “It is impossible that the pedestrian light A and the car traffic light B are green at the same time.”, “When the plane’s speed is greater than 400 km/h, then the flaps must not be extended.”, or “A warning light has to be lit in the plane’s cockpit if the last weather forecast received by radio is older than 30 minutes.”. These properties are also given in a formal language with an unambigous semantics.

Given the formal behaviour description and formulas expressing a required property, algorithms check whether the system satisfies the property: First, they generate a complete set of all configurations (“*states*”) that may be reached by the system’s description, including information on which states may succeed a given state. Then, all possible sequences of states are checked to see whether the property is true for them or not. This procedure is repeated for every property. Thus, model checking verifies firmly and, most notably, *exhaustively* that the system behaves as it should.

1.1.2 Abstract modelling

Data, concurrency, and real time. Textual languages and graphical models that are used for system descriptions in model checking must be simple enough to enable the use of efficient verification algorithms. However, when the objective is to perform formal verification of *realistic* systems, simple languages are not appropriate to model these systems.

For the concerns of this thesis, the term “realistic” refers to three aspects that must be provided in a suitable language: complex data structures, concurrency, and activity in real time. These three aspects are discussed below.

To handle complex *data* structures, a modelling language must cover both representation and manipulation of data:

- *Representation:* Simple data types (such as booleans, integers, and enumerated types) and structured data types (such as arrays, records, lists, unions, sets, and trees) can occur as parameters in systems we wish to model. Therefore, a suitable language must offer the possibility of user-defined data types.

  For instance, a system describing a mobile phone mast controller might have a data type *client* defined as an array containing phone number, service provider, etc., and another data type *connections*, defined as a set of *client* arrays.

- *Manipulation:* Mathematical functions need to be represented. These may be predefined for standard data types (e.g., addition of integers), but obviously, user-defined types only make sense when functions also can be user-defined.

  For instance, given a variable *current_clients* of type *connections*, update functions are necessary to represent new clients entering or current clients leaving the range of the mast.

To handle *concurrency* i.e., to describe systems that are composed of two or more independent subsystems (which we call *processes* in the following), a modelling language must cover communication between processes and process activation and deactivation.

- *Communication:* When several processes are executing concurrently, interaction between them has to be possible. For instance, in a system describing a plane, a *cockpit* process sends a message to the *flaps* process to order their extension.

  A communication between processes can be asynchronous (i.e., one process sends a message, then one or several other processes receive the message) or *synchronized* (i.e., all processes involved in the communication participate simultaneously). For synchronous communication, we do not need to identify the sender and the receivers.

- *Activation/Deactivation:* During the execution of a system, the set of concurrent processes is not necessarily static: New processes may be created, old ones may disappear.

  For instance, the arrival and the departure of different mobile phone users in the range of the mobile phone mast can be represented by the activation and deactivation of several processes, each of which corresponds to one user.

A model of a process expresses the order in which it performs its communications. However, sometimes such a *qualitative* description is not sufficient, and it is necessary to provide *quantitative* information on how much time elapses during the execution. To handle *real time*, a modelling language must cover delays, urgency, and latency.

- *Delays:* A suitable language must be able to express inaction of a process during some time. This is usually an abstract representation of an activity that takes time to be executed, like a physical displacement e.g., the time it takes to extend the flaps.

- *Urgency:* When a process is ready to perform a certain communication, it can make sense to force the immediate execution of this communication, instead of allowing time to elapse.

  For instance, when a mobile phone receives a message from the mobile phone mast process that there is an incoming call, then the ring tone is played immediately.

- *Latency:* A further refinement of this idea is to indicate that a limited amount of time can elapse before a communication becomes urgent.

  In the previous example, it could take the mobile phone process an indefinite but very limited time to decide which ring tone to play.

Complex data structures, concurrency, and activity in real time are not independent aspects of a system. Combining these aspects requires that we consider additional concepts:

- *Transmission of values:* In a language that combines data and concurrency, it must be possible to express that data generated in one process can be used in another process.

  For instance, the process representing the airspeed indicator of a plane must transmit the value of the current speed to the process representing the autopilot.

- *Timeout:* In a language that combines time and concurrency, communications may depend on the elapsing of time. When the system allows for two or more processes to communicate by a synchronization, the model should be able to express that once a process is ready to perform the synchronization, it does not wait arbitrarily, but becomes unavailable for the communication after a certain amount of time.

  For instance, after the mobile phone mast has sent a message of an incoming call to a mobile phone, it only waits one minute for the mobile phone to accept this call.

The need for intermediate models. The description and verification of systems covering data, concurrency, and time has been a very active research subject for already more than two decades. In the late 1980s and the 1990s, many approaches belonged to one of the two following categories:

- *High-level languages* provide a purely textual and highly expressive syntax that enables very concise descriptions. Most notably, *process algebras* [97], already providing concise representations of complex data and concurrency, were extended with real-time constructs. Process algebras describe the behaviour of a system as an ordering of *communication events*, representing how the system is perceived externally, while the internal behaviour is abstracted from. Such atomic events are combined with mathematical operators e.g., expressing that two things happen one after another (sequential composition) or that two behaviours execute independently (parallel composition).

  Among the process algebras extended with real-time constructs are ACP/SAT [10], ATP [100], CCS with time [125], ET-Lotos [91], μCRL [29], RT-Lotos [47], Timed CSP [108, 50, 103], Timed Lotos [31]. Various aspects from these languages converged into the E-Lotos language standardized by Iso [83].

- *Graphical models* on the other hand combine textual descriptions with a visual component, thus enabling a more intuitive application. Most notably, models like automata networks and Petri nets that already provide intuitive representations of concurrency were extended with data and real-time constructs. The most famous of these models are *timed automata* [4] and *time Petri nets* [95].

  Based on simpler syntax and semantics rules than the process algebras mentioned above, such graphical models enabled the development of successful algorithms and software tools for simulation and formal verification.

Experiences with these two approaches in the modelling of systems with data, concurrency, and time, revealed complementary strengths and weaknesses:

- High-level languages are very expressive and concise, but their elaborate structure makes formal verification tools hard to design. In some cases, even the formal description of their semantics becomes very complex.

- Graphical models serve as input languages for several verification tools, but are of limited capability for concise descriptions: The representation of a complex system can easily become unstructured and unreadable due to a large number of overlapping edges.

To overcome these problems, *intermediate models* have been developed during the last few years, with a design inspired from both approaches and aiming to combine the strengths of each. Ideally, it is then possible to specify a system in a high-level language, then to translate the specification into an intermediate model, and finally to translate this to a graphical model, where tools can be applied to perform model-checking, simulation, etc. The major difficulty in such a translation chain is preservation of semantics: One can use the results from the graphical model level to reason about the initial specification only if the semantics of the initial specification are preserved.

A proposition for a new intermediate format. The purpose of this thesis is the introduction of the new intermediate model ATLANTIF (*Asynchronous Timed Language Amplifying* NTIF). Although several propositions for intermediate models have been made during the last few years, all of them lack important constructs required to support recent high-level languages:

- Several industrial models based on model-driven tool developement (e.g., AADL [57], SYSML [72], and UML/MARTE [56]) have only a semi-formal definition. Therefore they are insuitable for exhaustive formal verification.

- Many approaches consisted of models for the efficient compilation of concurrent systems with data, yet without real-time constructs. For instance, the OPEN/CAESAR framework [60] in the CADP toolbox enables formal verification and simulation for models in different input formats. Similarly, the NIPS compiler [124] translates from different high-level modelling languages to a byte-code model enabling model checking.

- Other models provide formal semantics and real-time constructs, but are limited to processes whose behaviour is defined by simple transitions instead of high-level constructs; for instance the ALTARICA [45] model and the BIP [11] model.

- The IF-2.0 [38] model does not provide communication by synchronization. Other models, such as MODEST [30], do not provide complex synchronization operators.

- Efforts have been made to integrate high-level constructs into sequential processes of intermediate models, for instance in the Ntif (*New Technology Intermediate Form*) [61] model. Partially based on Ntif, the Fiacre [17] model also provides concurrency and real-time constructs.

  However, the real-time constructs provided by Fiacre only can express time constraints on a global level i.e., related to a synchronization among several processes that are ready to communicate. High-level languages like E-Lotos, on the other hand, can express time constraints on a local level i.e., processes themselves may have a limited availability for synchronizations.

Our objective for the definition of Atlantif is to overcome these restrictions and to provide translations to verification tools.

1.2 Overview of this thesis

Chapter 2 will fix the conventions of the notation that is used in this thesis. The remainder consists of two main parts, described in the following.

Definition of ATLANTIF. Chapters 3 and 4 describe the design of Atlantif.

The basis for this design is established in Chapter 3, which presents and analyzes existing high-level languages, graphical models, and intermediate models along with tool implementations. A focus will lie on the choices that have to be made when defining a new model.

Based on this analysis, Chapter 4 formally defines the syntax and semantics of Atlantif. This new model is based on Ntif: The definition of untimed sequential processes with complex data handling is essentially identical.

Atlantif extends Ntif with real-time constructs, expressing delays and time restrictions in processes as well as urgency. Atlantif also introduces concurrency constructs, which enable the parallel execution and synchronization of processes.

Application of ATLANTIF as intermediate format. Chapters 5, 6, and 7 show how Atlantif can indeed be used as an intermediate model. They describe connections from high-level languages to Atlantif and from Atlantif to graphical models.

Chapter 5 lists several examples of constructs typically occurring in high-level languages (e.g., exception handling and asynchronous termination) and shows how each of them can also be represented in Atlantif.

Chapter 6 presents two translations from subsets of Atlantif to other models, timed automata (Uppaal) and time Petri nets (Tina).

As an application of Chapters 5 and 6, Chapter 7 provides an example: We model in Atlantif a lift and demonstrate how this specification is translated into a Tina time

Petri net on which model checking is performed.

The conclusion in Chapter 8 compares ATLANTIF with other intermediate models, underlining that the range of constructs it provides is not covered by any of them, and thus showing that ATLANTIF is indeed a contribution for the high-level modelling of systems. We discuss how ATLANTIF extends the class of systems that can practically be formally verified, and end with some perspectives for the future of ATLANTIF.

The appendixes contain some additional algorithms and proofs (Appendix A), additional examples (Appendix B), the complete syntax of ATLANTIF as it is implemented in our tool (Appendix C), and an extended summary of this thesis in French (Appendix D).

Chapter 2

Notation

The following notation is used throughout this thesis, particularly Chapters 4 and 6. Most of this notation is in common usage.

- To distinguish the definition of a specific set, a function, etc., we use the notation $X \stackrel{\text{def}}{=} Y$, where the object named X is defined by the term Y.

- $1..n$ and $0..n$ with $n \in \mathbb{N}$ are shorthand notations for the sets $\{i \in \mathbb{N} \mid 1 \leq i \leq n\}$ and $\{i \in \mathbb{N} \mid i \leq n\}$ respectively. Note that $0 \in \mathbb{N}$.

- Let $\varphi, \varphi' : X \to X'$ be partial functions:

  - $\mathit{dom}(\varphi)$ denotes the *domain* of φ i.e., the subset of X on which φ is defined.
  - $\mathit{image}(\varphi), \mathit{image}(\varphi')$ are the *images* of these function i.e., the subsets of X' in which they take values.
  - We define the *update* operator $\oslash$ and the *restriction* operator $\ominus$ for elements $x_1, \ldots, x_n \in X$ by:

$$\begin{aligned}\varphi \oslash \varphi' \stackrel{\text{def}}{=} \varphi'' \text{ where } \varphi'' : X \to X', \mathit{dom}(\varphi'') = \mathit{dom}(\varphi) \cup \mathit{dom}(\varphi'),\\ \text{and } (\forall x \in \mathit{dom}(\varphi'')) \ \varphi''(x) = \begin{cases} \varphi'(x) & \text{if } x \in \mathit{dom}(\varphi') \\ \varphi(x) & \text{otherwise} \end{cases}\end{aligned}$$

$$\begin{aligned}\varphi \ominus \{x_1, \ldots, x_n\} \stackrel{\text{def}}{=} \varphi'' \text{ where } \varphi'' : X \to X', \mathit{dom}(\varphi'') = \mathit{dom}(\varphi) \setminus \{x_1, \ldots, x_n\},\\ \text{and } (\forall x \in \mathit{dom}(\varphi'')) \ \varphi''(x) = \varphi(x)\end{aligned}$$

  - If the domain of φ is a finite set $\{y_1, \ldots, y_n\}$ and $\varphi(y_1) = y'_1, \ldots, \varphi(y_n) = y'_n$, then we also may write φ in the form $[y_1 \mapsto y'_1, \ldots, y_n \mapsto y'_n]$. If the domain of φ is empty, then we also may write it $\varnothing$.

- $\mathbb{Q}_{\geq 0}$ denotes the set of non-negative rationals and $\mathbb{R}_{\geq 0}$ denotes the set of non-negative reals.

- For arbitrary expressions E_1, E_2, and a variable V, we note $[E_1/V]E_2$ for the expression that is obtained when each occurrence of V in E_2 is replaced by E_1.
- For a set X, $card(X)$ denotes the number of elements (or *cardinality*) of X. In the context of this work, this only applies to finite sets.
- For a set X, $\mathcal{P}(X)$ denotes the power set of X i.e., the set containing all subsets of X.

Many definitions are given throughout this thesis. To allow a fast access, the locations of these definitions are given in the following tables:

Function	*Definition Page*
accept	60
act	61
tl_off	140
label	86
new_guard	140
new_limit	141
reach	214
sense	136
shift	29
tag	75
trans	139
$trans_p$	156
transitions_and_locations	138
type (on expressions)	56
type (on offers)	60
type (on patterns)	58
ucuns	214
update_exp	140
update_exp'	155
θ	81
π	81
ρ	57
$\ominus$	9
$\oslash$	9
$+$ (on labels)	69
$+$ (on $(\mathbb{U} \to \mathbb{D}) \times \mathbb{D}$)	86

Table 2.1: List of function definitions

Set	*Definition Page*
$accessible(V)$	75
$\mathbb{D}$	16
$decl(u)$	60
$def(O)$	60
$def(P)$	58
$\mathbb{G}$	55
$\mathbb{L}_1$	61
$\mathbb{L}_1'$	81
$\mathbb{L}_2$	81
$match(v, \rho, P)$	59
$read(A)$	61
$read(E)$	57
$read(O)$	60
$read(P)$	59
$read(u)$	72
$read(W)$	65
$shadow(\mathcal{U})$	215
$start(G)$	76
$stop(G)$	76
$sync(G)$	78
$\mathcal{S}, \mathcal{S}_u$	55
$\mathbb{S}$	81
$\mathbb{T}$	81
$use(E)$	57
$use(O)$	60
$use(P)$	58
$\mathcal{U}$	55
$\mathcal{U}_0$	79
$\mathbb{U}$	55
Val	57
$write(A)$	61
$write(O)$	60
$write(P)$	59
$write(u)$	72
$0..n$, $1..n$	9

Table 2.2: List of set definitions

Predicate	*Definition Page*
$emission(G, i, u_0)$	136
$enabled(S, l, \mu, S')$	85
$eval(E, \rho, v)$	57
$next\_\alpha(\alpha, \mathcal{U}, G, \alpha')$	85
$next\_\theta(\theta, \mathcal{U}, t_0, G, \theta')$	84
$next\_\pi(\pi, \pi_1, G, \pi')$	84
$next\_\rho(\rho, \rho_1, \mathcal{U}, G, \rho')$	85
$relaxed(S)$	85
$synchronizing((S, \alpha), l, \mu, (S', \alpha'))$	83
$up\_lim(Q, D, t)$	67
$validity\_stable(G)$	77
$validity\_stable'(G)$	215
$valid\_active(\mathcal{U})$	77
$win\_eval(W, \rho, D)$	66
$\stackrel{\text{def}}{=}$	9
$(A, d, \rho) \stackrel{l}{\Longrightarrow} \sigma$	62
$\succ$	73
$\succeq$	73

Table 2.3: List of predicate definitions

Term	*Definition Page*
binary synchronization	25
blocking condition	62
clock	23
chain	82
dense time	17
discrete time	17
discrete transition	17
execution path	72
graphical model	22
high-level language	33
intermediate model	37
labelled transition system	16
local state	62
maximal progress of urgent actions	18
modality	65
phase	62
run	19
store	57
strong deadline	46
time additivity	18
time determinism	18
time domain	16
time Petri net	28
time window	65
timed automaton	23
timed labelled transition system	17
timed transition	17
timing option	79
timelock	19
unit affection	83
validity-stability	77
weak deadline	46
well-binding	59
well-activatedness	80
well-accessible	74

Table 2.4: List of terms introduced

Dynamic semantics rule	*Definition Page*
$(assign_d)$	62
$(assign_n)$	63
$(case)$	69
$(comm)$	67
(if_1)	70
(if_2)	70
$(null)$	72
(rdv)	85
$(reset)$	64
$(select)$	71
(seq_1)	69
(seq_2)	69
$(stop)$	68
$(time)$	86
(to)	67
$(wait)$	64
$(while_1)$	71
$(while_2)$	71
$(\varepsilon\text{-}elim)$	73

Table 2.5: List of semantic rules

Chapter 3

Overview and classification of formal models

Abstract To be of use, an intermediate model should have an expressive power that covers the most important constructs of commonly used high-level languages, while having structural similarities to lower-level models, making them accessible for translations from the intermediate model. Therefore, this chapter provides a detailed analysis of existing formats, which starts with presenting common semantic models, then discusses the most important graphical models, several high-level languages, and finally several intermediate models. A comparison of these formats at the end of this chapter concludes that the definition of a new intermediate format is justified.

In this chapter, we make a detailed analysis of existing formal models and languages that combine expressive power regarding data, concurrency, and real time.

We begin by discussing semantic models i.e., simple models used by other models and languages to define their formal semantics, thus providing a basis for further comparisons.

We then discuss graphical models, which are situated on a higher level of abstraction. We define two such models, namely timed automata and time Petri nets, give an overview of their dialects, and then mention further graphical models. We also list various software tools that have been developed to perform formal verification on the models presented.

Then, we proceed to high-level languages, by presenting different families of process algebras, which provide more concise notations than graphical models. Only a few software tools are available to perform formal verification directly on models written in high-level languages.

With the differences between graphical models and high-level languages identified, we then explain the need for intermediate models i.e., models using structures and constructs of graphical models as well as of high-level languages and situated on an intermediate abstraction level. We then present several existing intermediate models and describe their expressive power.

The conclusion of this chapter gives an overview of criteria that can be used to compare the different models and languages. These criteria represent the choices that have to be made when defining a new model. Against the background of these choices, we will see that several combinations of constructs are not provided by any of the existing intermediate models, and that therefore the definition of a new intermediate format is justified. Chapter 4, which gives such a definition, takes the choices identified here as a basis for the new intermediate format Atlantif.

3.1 Semantic models

3.1.1 Timed labelled transition systems

Definitions

This section provides the definition of the timed labelled transition system (Tlts) model, which most other formalisms in the remainder of this chapter will use to define their semantics. Thus, it represents a simple, language-independent model and a common basis for a reasonable comparison between different formal models.

The Tlts model extends the untimed "labelled transition systems" (Lts) by including information on time elapsing. We develop the definition through three steps: First, we formally define the LTS model, which itself is used as a semantic model of untimed systems. Second, we define a "time domain" structure, which will be used to quantify time. Third, we combine those two definitions to obtain the Tlts model.

Definition 3.1. *A* labelled transition system (Lts) *is a 4-tuple* $(\Sigma, A, \rightarrow, S_0)$*, defined as follows:*

- Σ *is a (possibly infinite) set of* states*, written* S, S', S_0, S_1*, etc.*
- $S_0 \in \Sigma$ *is called the* initial state*.*
- A *is a (possibly infinite) set of* discrete labels*, written* a, a', a_0, a_1*, etc. It contains one special element written* τ*, called the* silent *label.*
- *The set of triples* $\rightarrow \subseteq (\Sigma \times A \times \Sigma)$ *is called the* transition relation*. Following common practice, we will write* $S \xrightarrow{a} S'$ *instead of* $(S, a, S') \in \rightarrow$*.*

Definition 3.2. *A* time domain *is a structure* $(\mathbb{D}, 0, <, +)$ *satisfying the following:*

- *The carrier set* $\mathbb{D}$ *is finite or infinite. We write* t, t', t_1, t_2*, etc. for its elements.*
- $<$ *is a total order on* $\mathbb{D}$*.*
- 0 *is a minimal element w.r.t. the order* $<$ *(i.e.,* $(\forall t \in \mathbb{D})\ 0 = t \lor 0 < t$*).*

- $+$ *is an associative and commutative operation, totally defined on* $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$*, and accepting* 0 *as neutral element.*
- *Each element can be complemented to each bigger element. Formally:*
$$(\forall t_1, t_2 \in \mathbb{D})\ (t_1 < t_2 \Leftrightarrow (\exists t_3 \in \mathbb{D})\ t_3 \neq 0 \wedge t_1 + t_3 = t_2)$$
- $<$ *is stable by constant addition. Formally:*
$$(\forall t, t')\ 0 < t \Rightarrow t' < t + t'$$

As it is common usage, the term time domain *will often simply refer to the carrier set* $\mathbb{D}$.

We say that the time domain is dense *if the carrier set is infinite and dense w.r.t.* $<$ *i.e.,* $(\forall t_1, t_2 \in \mathbb{D})\ t_1 < t_2 \Rightarrow (\exists t_3 \in \mathbb{D})\ t_1 < t_3 < t_2$. *We say that the time domain is* discrete *if the carrier set is discrete w.r.t* $<$ *i.e.,* $(\forall t_1 \in \mathbb{D})\ (\exists t_2 \in \mathbb{D})\ t_1 + t_2$ *is the smallest element greater than* t_1 *(thus, each element has a direct successor).*

Clearly, being discrete and being dense are mutually exclusive. Examples for dense time domains are the non-negative rationals $\mathbb{Q}_{\geq 0}$ and the non-negative reals $\mathbb{R}_{\geq 0}$. An example for a discrete time domain are the natural integers $(\mathbb{N}, 0, <, +)$; all other discrete time domains are isomorphic to this structure. The set $\{0\}$ is neither discrete nor dense; obviously this is the only possible example where $\mathbb{D}$ is finite.

Definition 3.3. *A* timed labelled transition system (TLTS) *is a 5-tuple* $(\Sigma, A, \mathbb{D}, \rightarrow, S_0)$*, defined as follows:*

- Σ*,* S_0*, and* A *are defined as in Definition 3.1 above.*
- $\mathbb{D}$ *is the carrier set of a time domain such that* $A \cap \mathbb{D} = \varnothing$*. We write* l, l', l_0, l_1*, etc. for the elements of* $A \cup (\mathbb{D} \setminus \{0\})$.
- *The set of triples* $\rightarrow\ \subseteq (\Sigma \times (A \cup (\mathbb{D} \setminus \{0\})) \times \Sigma)$ *is called the* transition relation. *We will also write* $S \xrightarrow{l} S'$ *for a triple* $(S, l, S') \in\ \rightarrow$*. If* $l \in \mathbb{D}$*, we call* (S, l, S') *a* timed transition*; if* $l \in A$ *we call it a* discrete transition[1].

Intuitively, a timed transition (S, t, S') represents the elapsing of t time units, and a discrete transition represents a communication action of the system, either hidden (if labelled τ) or visible (otherwise).

For given $S \in \Sigma, l \in A \cup (\mathbb{D} \setminus \{0\})$, we may use the shorthand $S \xrightarrow{l}$ instead of writing $(\exists S' \in \Sigma)\ S \xrightarrow{l} S'$.

By the condition $A \cap \mathbb{D} = \varnothing$, it is assured that each transition is either timed or discrete, not both. Thus, we impose the assumption that discrete transitions have a duration of zero

[1]Note that we use the word "discrete" at the same time to distinguish time domains (which can either be dense or discrete) and to distinguish transitions in TLTS (which can either be timed or discrete), because in both cases it is the commonly used term. For a given TLTS, the time domain is fixed, therefore it is always clear by the context in which sense "discrete" is used.

time units. In [100], it has been argued that this assumption does not go against generality and that it is a useful simplification. [98] even uses a quantum-mechanical argument in favor of this assumption, by stating that discrete transitions represent energy changes of a system, which cannot be measured simultaneously with time.

Note that in the literature, a TLTS is sometimes also called *timed transition system* (TTS) or *labelled timed transition system*. Note also that if we choose a time domain such that $\mathbb{D} = \{0\}$, then for practical purposes the definition of a TLTS coincides with the definition of an LTS.

Properties of TLTSs

In the literature concerning the TLTS model, different authors (e.g., [99, 91]) describe three properties of well-timedness that a TLTS should satisfy to be intuitively consistent:

Definition 3.4. *A* TLTS $(\Sigma, L, \mathbb{D}, \rightarrow, S_0)$ *is called* well-timed, *if and only if the following three conditions are all true:*

1. Time additivity*: Two succeeding timed transitions are equal to their sum. Formally, for each* $S_1, S_2 \in \Sigma, t_1, t_2 \in (\mathbb{D} \setminus \{0\})$*:*
$$S_1 \xrightarrow{t_1+t_2} S_2 \textit{ iff } (\exists S_3 \in \Sigma)\ S_1 \xrightarrow{t_1} S_3 \xrightarrow{t_2} S_2$$
2. Time determinism*: From a given state, the elapsing of a given duration cannot lead to different states. Formally, for each* $S_1, S_2, S_3 \in \Sigma, t \in (\mathbb{D} \setminus \{0\})$*:*
$$\textit{if } S_1 \xrightarrow{t} S_2 \textit{ and } S_1 \xrightarrow{t} S_3,\ \textit{then } S_2 = S_3$$
3. Maximal progress of urgent actions*: Assuming a set* $U \subseteq A$ *of labels that are called* urgent*. A state allowing a discrete transition with an urgent label must not allow a timed transition. Formally, for each* $S_1 \in \Sigma, l \in A, t \in (\mathbb{D} \setminus \{0\})$*:*
$$\textit{if } l \in U \textit{ and } S_1 \xrightarrow{l}\ ,\ \textit{then } \neg\left(S_1 \xrightarrow{t}\right)$$

Intuitively, time additivity expresses that time progress is independent of observation. Time determinism expresses that time can only advance in one direction. More technically, the mere elapsing of time does not put constraints on which discrete actions may follow i.e., it does not resolve any choices. Maximal progress often supposes that the set of urgent (i.e., undelayable) actions corresponds to the singleton containing the hidden action τ. Our semantics definition in Chapter 4 will use a different set.

Note that the combination, the definition details and the naming of well-timedness properties are not entirely consistent in the literature e.g., time additivity is called *stutter closure* in [75], and it is described as two properties in [20].

Other interesting properties of TLTSs have been defined. An important one is the so-called *non zenoness*, which is discussed in the literature with many variations in terminology and formal definition, for instance in [75], in [100] (called *well-timedness* there), or in [90].

Definition 3.5. (i) *A* run *in a* TLTS $(\Sigma, A, \mathbb{D}, \rightarrow, S_0)$ *is an infinite sequence of elements of* Σ *alternating with elements of* $\rightarrow$*, such that it begins with* S_0 *and it has the form* $S_0, (S_0, l_0, S_1), S_1, (S_1, l_1, S_2), S_2, \ldots$ *(with* $S_1, S_2, \ldots \in \Sigma, l_0, l_1, \ldots \in A \cup \mathbb{D}$*). A* finite run *is the prefix of a run, such that this prefix ends with an element of* Σ.

(ii) *The duration of a run or a finite run is the sum of all timed labels occurring in it. A run is* diverging *if this sum tends to infinity.*

(iii) *A* TLTS *is called* zeno *if it allows a finite run which is not the prefix of a diverging run. Otherwise, it is called* non zeno.

Intuitively, *non zeno* means that the system represented by the TLTS may always continue in a way where time may elapse indefinitely. A zeno behaviour can be caused by two different reasons:

1. The TLTS contains a state that does not have any outgoing transitions. Such a state is called a *timelock* state.
2. The TLTS contains a state the runs starting from which have a finite duration (either because they only contain a finite number of timed transitions, or because their sum does not diverge). Such a case is shown in Fig. 3.1: There, only one run is possible and its duration is one time unit; but no finite prefix can reach this duration. Clearly, such a TLTS describes an unrealistic behaviour.

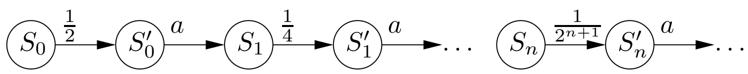

Figure 3.1: TLTS with zeno behaviour

This definition is a rather general variation of the definitions for zeno behaviour that can be found in the literature. For our analysis in the following chapter, a tighter definition will therefore be useful (where a *reachable* state is a state that is contained in at least one run):

Definition 3.6. *We suppose a dense time domain.*

A TLTS *is called* strict zeno *if there is a* $t \in \mathbb{D}$ *and a reachable state* S *satisfying all of the following:*

- *There is a run* $S_0, (S_0, l_0, S_1), S_1, \ldots$ *with* $S = S_n$ *for some* $n \in \mathbb{N}$*, such that for all* $i \geq n$*,* l_i *is a timed label and* $\sum_{i \geq n} l_i = t$.
- *For each run* $S_0, (S_0, l_0, S_1), S_1, \ldots$ *with* $S = S_n$ *for some* $n \in \mathbb{N}$*: For all* $i \geq n$*,* l_i *is a timed label and* $\sum_{i \geq n} l_i \leq t$ *(i.e., the maximal duration after* S *is limited by* t*).*
- *There is no finite run* $S_0, (S_0, l_0, S_1), \ldots, S_m$ *with* $S_n = S$ *for some* $n < m$ *and* $\sum_{n \leq i < m} l_i = t$ *(i.e., the* t *time units cannot elapse in finitely many steps).*

Intuitively, the runs that are decisive for the definition of strict zeno are those that approach infinitesimally close the bound t without being able to reach it by a finite number of steps. Note that TLTSs with timelock states are not necessarily strict zeno, nor is the TLTS from Fig. 3.1 strict zeno. Thus, zeno does not imply strict zeno. On the other hand, strict zeno clearly implies zeno.

$$S_0 \xrightarrow{\frac{1}{2}} S_1 \xrightarrow{\frac{1}{4}} S_2 \xrightarrow{\frac{1}{8}} \dots \quad S_n \xrightarrow{\frac{1}{2^{n+1}}} \dots$$

Figure 3.2: Strict zeno TLTS

A simple example for a strict zeno TLTS is shown in Fig. 3.2. The duration of the only possible run in this example is $\Sigma_{n\geq 1}\frac{1}{2^n} = 1$, but no finite run of duration 1 is possible. Note that the satisfaction of time additivity does not prohibit strict zenoness: If we extend the TLTS of Fig. 3.2 to its time additive closure (i.e., we add the minimum of additional states and transitions such that it becomes time additive), it still does not contain a transition from S_0 labelled by 1, because the criterion of time additivity is restricted to sums of two transitions. By multiple application, this extends to any finite number of transitions, but not to infinitely many.

Differences between discrete and dense time domains. A TLTS with a discrete time domain behaves differently from one with a dense time domain. Discrete time is obviously easier to define, but it is also less intuitive, as it opposes the human perception of time in reality that can be arbitrarily divided[2]. Note that discrete time can be adjusted in its granularity: A run $S_0 \overset{\frac{1}{3}}{\to} S_1 \overset{a}{\to} S_2 \overset{\frac{2}{3}}{\to} S_3 \overset{a}{\to} S_4 \overset{\frac{1}{3}}{\to} S_5 \overset{a}{\to} S_6 \overset{\frac{2}{3}}{\to} \dots$ is not necessarily from a TLTS with a dense time domain like $\mathbb{Q}_{\geq 0}$, it can also appear in a TLTS with a discrete time domain $\{0, \frac{1}{3}, \frac{2}{3}, 1, 1\frac{1}{3}, \dots\}$. Nevertheless, not every system can be "discretized" in this way, as shown in [41]. Thus discrete time domains are strictly less expressive than dense time domains.

A detailed comparison between discrete and dense time domains can be found in [39].

Differences between $\mathbb{Q}_{\geq 0}$ and $\mathbb{R}_{\geq 0}$. Regarding dense time domains, both $\mathbb{Q}_{\geq 0}$ and $\mathbb{R}_{\geq 0}$ are used in the literature. From a practical point of view however, there is almost no difference:

- Neither set is "denser" than the other, because $\mathbb{Q}_{\geq 0}$ lies dense in $\mathbb{R}_{\geq 0}$ and vice versa i.e., for all $x, y \in \mathbb{R}_{\geq 0}$ with $x < y$, there is a $z \in \mathbb{Q}_{\geq 0}$ such that $x < z < y$ (and conversely).
- Moreover, model theory shows that $(\mathbb{Q}_{\geq 0}, 0, <, +)$ and $(\mathbb{R}_{\geq 0}, 0, <, +)$ satisfy *elementary equivalence* i.e., each first order logic formula satisfied in one of these structures

[2] Note that this opinion is not unopposed e.g., [98] states discrete time to be more intuitive, because system states in a computer change discretely.

is also satisfied in the other[3].

Timed properties discussed in formal verification are normally expressed in first order logic formulas, extended with additional constants. In theory, a formula in the real numbers structure could express for instance that "precisely $\sqrt{2}$ time units elapse" (such examples are discussed in [90]). In practice however, we talk about automated formal methods, and computers have no means to efficiently represent irrational numbers[4]; therefore this difference is without real impact for us.

Model checking on TLTS

The semantic model (e.g., in form of a TLTS) of a formal specification can be the basis for model checking, which consists in verifying that a formula given in a temporal logic, such as μ-calculus, CTL, or LTL, is satisfied by the semantic model of the specification. Although the cited temporal logics are named "temporal", they do not express real-time aspects such as the precise time that elapses between two events, but merely the *order* (relative to time elapsing) of events. Therefore, timed extensions to temporal logics have been defined, such as TCTL (*timed* CTL) [3] which also provide constructs to express how much time may or may not elapse between events.

The model checking algorithms defined for TLTSs differ from those defined for LTSs: If the time domain is infinite, the TLTS is usually, but not necessarily, of infinite size (if the time domain is dense, then only TLTSs without timed transitions can be smaller than infinite). Thus, an algorithm has to use symbolic techniques that enable the representation of a TLTS as a finite structure.

Some of the syntax restrictions that occur in the graphical models (cf. Section 3.2) are due to the fact that these models have been developed along with verification algorithms that use such symbolic representations (e.g., [3]).

3.1.2 Alternative semantic models

TLTSs are a natural extension of LTSs. Although they are the most frequently used semantic model for timed systems, alternative approaches exist.

Labels with actions and absolute time stamps. The timed model proposed in [9, 85] extends the LTS model by transitions labelled with expressions of the form $a(t)$, where a is a (discrete) label, and t a time value. Time is represented in an absolute way i.e., the time value expresses how much time has passed between the beginning of the run and this occurrence of a.

[3]Elementary equivalence is a less strict relation than isomorphism, the latter obviously not being satisfied.

[4]Even if some programming languages denote them as "reals", floating point numbers are always rationals.

Fig. 3.3 *(ii)* shows such a semantic model, beside the identically behaving TLTS model in *(i)*.

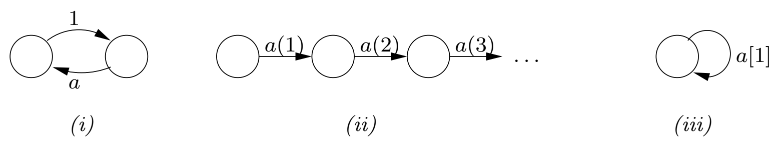

Figure 3.3: Different LTS extensions (time domain $\mathbb{N}$)

Labels with actions and relative time stamps. A similar approach [49] uses relative values instead of absolute values. Transitions are also labelled by expressions of the form "$a(t)$", where a is a (discrete) label, and t a time value, but here, t expresses how much time has passed between the last transition and the current occurrence of a. This model is illustrated in Fig. 3.3 *(iii)*.

Note that in a model using labels with action and time stamp (absolute or relative), the properties of time additivity and time determinism (cf. Definition 3.4) are automatically satisfied, because there are no isolated timed transitions.

Relations between different semantic models. As the last remark indicates, models using labels with actions and time stamps can only define graphs that satisfy time additivity and time determinism. Thus, not everything that can be expressed in the TLTS model can be expressed in the alternative models. On the other hand, given that a TLTS that does not satisfy time additivity and time determinism lacks intuition, this difference in the expressive power can be seen as irrelevant.

More generally, a translation from labelled transition systems with absolute or relative time stamps into a corresponding TLTS is always possible, simply by splitting up each time stamp transition into one timed transition and one discrete transition (or only one discrete transition if no time elapses). The inverse is not true; for instance, a TLTS only containing timed transitions has obviously no correspondent in a time stamp model.

3.2 Graphical models

3.2.1 Overview

In this work the term *graphical models* will refer to models that have been traditionally defined using a *finite* graphical notation (despite their possibly infinite behaviour). This finite representation is realized using *symbolic* notations, which represent sets of possible values e.g., by a formula that imposes constraints on the value of a variable. Such notations are in particular useful to represent real-time behaviour: For instance, when an arbitrary amount of time may elapse, this amount could be represented symbolically on a single transition by an interval $[0, \infty[$; a semantic model would instead define an infinite number of transitions. Moreover, graphical models have formally defined and simple semantics.

Graphical models are appropriate to model simple systems. Their strength is the existence of algorithms that transform graphical models into symbolic representations of TLTSs [19, 5], which enables model checking to be performed.

In this section, we will discuss two main types of graphical models, namely timed extensions of the automata model and timed extensions of the Petri net model. Several other models, all of which have been developed along with one specific verification tool, will also be discussed briefly.

3.2.2 Timed automata

Formal definition

The network of timed automata model is used in several tools for simulation and formal verification, such as UPPAAL [89], RED [123], SGM [80], CMC [88], KRONOS [126], or RABBIT [27].

Timed automata were originally defined by Alur and Dill [4, 5] and appeared later in the literature in a large number of variations (such as [75, 12]). The following definitions represent a common standard among the different definitions.

Definition 3.7. (i) *A* clock X *is a variable that has values ranging in* $\mathbb{R}_{\geq 0}$.

(ii) *A* clock constraint δ *is an expression recursively defined by one of the following forms: "$X \leq c$", "$c \leq X$", "$X - Y \leq c$", "$c \leq X - Y$", "$\neg\delta$", or "$\delta_1 \wedge \delta_2$", where $c \in \mathbb{N}$ and X, Y are clocks. For a set $\mathcal{X}$ of clocks, we write $\Phi_{\mathcal{X}}$ the set of clock constraints built with clocks from $\mathcal{X}$.*

Clocks are used to represent the elapsing of time: All clocks then evolve continously and with a constant speed that is the same for all clocks.

Definition 3.8. *A* timed automaton *(TA) is a 6-tuple $(\mathcal{X}, A, L, E, inv, l_0)$, where:*

- *$\mathcal{X}$ is a set of clocks.*
- *A is a finite set of* labels.
- *(L, E) is a finite directed graph, where the elements of the set L of nodes are called* locations[5] *and the elements of the set E of edges are called* transitions. *Each such transition is a 5-tuple $e = (l_1, \delta, a, \{X_1, \ldots, X_n\}, l_2)$, where*
  - $l_1, l_2 \in L$

[5]When we speak of "locations", we already apply a vocabulary oriented towards the translation to the UPPAAL tool (cf. Section 6.1). Although in other dialects, locations are called, for instance, "*modes*" (in the tools RED and SGM) or "*nodes*" (in CMC), UPPAAL is not the only dialect using the term "locations". For the remainder of Section 3.2.2, we will continue to use UPPAAL vocabulary.

  - *the formula* $\delta \in \Phi_{\mathcal{X}}$ *is called the* guard
  - $a \in A$
  - $\{X_1, \ldots, X_n\} \subseteq \mathcal{X}$ *is called the* clock reset

- $inv : L \to \Phi_{\mathcal{X}}$ *is a function distributing an* invariant *to each location.*

- l_0 *is the* initial location.

Semantics. Given a set $\mathcal{X}$ of clocks, a function $\mu : \mathcal{X} \to \mathbb{R}_{\geq 0}$ is called a *clock interpretation*. A clock constraint $\delta \in \Phi_{\mathcal{X}}$ is *satisfied by* μ, if the expression obtained by replacing in δ each clock variable X by the value $\mu(X)$ evaluates to **true**.

The semantics of a timed automaton $(\mathcal{X}, A, L, E, inv, l_0)$ is defined by a TLTS $(\Sigma, A, \mathbb{R}_{\geq 0}, \to, (l_0, \mu_0))$, where $\Sigma, \to$ are the smallest sets satisfying:

- The states of Σ are tuples of the form (l, μ), where $l \in L$ and μ is a clock interpretation on $\mathcal{X}$.

- The initial state is $(l_0, \mu_0) \in \Sigma$, where μ_0 is the function that constantly returns zero.

- *Timed transition*: If $(l, \mu) \in \Sigma$, $t \in \mathbb{R}_{\geq 0}$, then $(l, \mu') \in \Sigma$ and $(l, \mu) \xrightarrow{t} (l, \mu')$ if the following are true:

  - μ' is a clock interpretation such that for each $X \in \mathcal{X}$, $\mu'(X) = \mu(X) + t$. We use the shorthand notation $\mu' = \mu + t$.
  - For each $t' \in]0, t]$, $inv(l)$ is satisfied by $\mu + t'$ i.e., the invariant formula of l holds during the entire time elapsing.

- *Discrete transition*: If $(l, \mu) \in \Sigma$, $(l, \delta, a, \{X_1, \ldots, X_n\}, l') \in E$, then $(l', \mu') \in \Sigma$ and $(l, \mu) \xrightarrow{a} (l', \mu')$ if the following are true:

  - δ is satisfied by μ.
  - μ' is a clock interpretation such that for each $X \in \mathcal{X}$, $\mu'(X) = 0$ if $X \in \{X_1, \ldots, X_n\}$ and $\mu'(X) = \mu(X)$ otherwise.
  - $inv(l')$ is satisfied by μ'.

Note that the original TA definition of [5] does not contain invariants. The reason for introducing invariants was to prevent a TA from idling indefinitely in one location. With this aim, [5] uses *Büchi* or *Muller acceptance conditions*, thus forcing a run to contain infinitely many discrete transitions. But this is a somehow artificial construction, thus less appropriate for modelling than using invariants.

Parallel composition. The semantics of a *network* of n timed automata

$$(\mathcal{X}_1, A, L_1, E_1, inv_1, l_1^0), \ldots, (\mathcal{X}_n, A, L_n, E_n, inv_1, l_n^0)$$

is defined by a TLTS $(\Sigma, A, \mathbb{R}_{\geq 0}, \rightarrow, ((l_1^0, \ldots, l_n^0), (\mu_1^0, \ldots, \mu_n^0)))$, where $\Sigma, \rightarrow$ are the smallest sets satisfying:

- The states of Σ are tuples of the form $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n))$, where $l_1 \in L_1, \ldots, l_n \in L_n$ and $\mu_1, \ldots, \mu_n$ are clock interpretations on the sets $\mathcal{X}_1, \ldots, \mathcal{X}_n$ respectively.
- The initial state is $((l_1^0, \ldots, l_n^0), (\mu_1^0, \ldots, \mu_n^0)) \in \Sigma$, where $\mu_1^0, \ldots, \mu_n^0$ are the functions that constantly return zero.
- *Timed transition*: Time can elapse in the composition if it can elapse in each single automaton. Formally:

  If $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \in \Sigma$, $t \in \mathbb{R}_{\geq 0}$, and $(\forall i \in 1..n)\ (l_i, \mu_i) \xrightarrow{t} (l_i, \mu_i')$, then $((l_1, \ldots, l_n), (\mu_1', \ldots, \mu_n')) \in \Sigma$ and $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \xrightarrow{t} ((l_1, \ldots, l_n), (\mu_1', \ldots, \mu_n'))$.
- *Discrete transition*: In the parallel composition of timed automata, two different kinds of discrete transition exist. The first kind corresponds to a discrete transition on the silent label $\tau \in A$ in a single automaton, while the others remain unchanged. Formally:

  If $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \in \Sigma$ and $(\exists i \in 1..n)\ (l_i, \mu_i) \xrightarrow{\tau} (l_i', \mu_i')$,
  then $((l_1, \ldots, l_i', \ldots, l_n), (\mu_1, \ldots, \mu_i', \ldots, \mu_n)) \in \Sigma$ and
  $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \xrightarrow{\tau} ((l_1, \ldots, l_i', \ldots, l_n), (\mu_1, \ldots, \mu_i', \ldots, \mu_n))$.

  The second kind of discrete transition is the *synchronization* among several timed automata. For the semantics of synchronizations, several different definitions exist in the literature, varying in particular about how many TAs synchronize. The three major approaches are the following:

  1. *Global discrete synchronization* [5]: **All** automata execute simultaneously the same action $a \in (A \setminus \{\tau\})$. Formally:

     If $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \in \Sigma$ and $(\forall i \in 1..n)\ (l_i, \mu_i) \xrightarrow{a} (l_i', \mu_i')$,
     then $((l_1', \ldots, l_n'), (\mu_1', \ldots, \mu_n')) \in \Sigma$ and
     $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \xrightarrow{a} ((l_1', \ldots, l_n'), (\mu_1', \ldots, \mu_n'))$.
  2. *Binary synchronization* [12]: The set of visible actions $(A \setminus \{\tau\})$ is composed of *actions* of the form "$!c$" and *co-actions* of the form "$?c$".

     In a synchronization, one automaton executes an action, and another automaton executes simultaneously the corresponding co-action, while the other automata remain unchanged. Formally:

     If $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \in \Sigma$
     and $(\exists i, j \in 1..n, i \neq j)\ (l_i, \mu_i) \xrightarrow{!c} (l_i', \mu_i') \wedge (l_j, \mu_j \xrightarrow{?c} (l_j', \mu_j')$,
     then $((l_1', \ldots, l_n'), (\mu_1', \ldots, \mu_n')) \in \Sigma$ (where $l_k' = l_k, \mu_k' = \mu_k$ for each $k \neq i, j$) and
     $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \xrightarrow{c} ((l_1', \ldots, l_n'), (\mu_1', \ldots, \mu_n'))$.

3. *Maximal event synchronization* [126]: The set of visible actions ($A \setminus \{\tau\}$) is the power set of a finite set $\mathcal{E}$ of *events* i.e., each transition in the timed automata is labelled by a set of one or several events. In a synchronization, several automata each execute one such event set simultaneously, where (i) every event must occur in at least two event sets, and (ii) the set of synchronizing automata must be maximal i.e., if an automaton can execute an event set containing an occurring event, then it must be among the synchronizing TAs. The other automata remain unchanged. Formally:

   If $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \in \Sigma,$ and
   $(\exists i_1, \ldots, i_m \in 1..n$ pairwise distinct$)$ $(\forall j \in 1..m)$ $(l_{i_j}, \mu_{i_j}) \xrightarrow{a_j} (l'_{i_j}, \mu'_{i_j}),$
   then $((l'_1, \ldots, l'_n), (\mu'_1, \ldots, \mu'_n)) \in \Sigma$ and
   $((l_1, \ldots, l_n), (\mu_1, \ldots, \mu_n)) \xrightarrow{\bigcup_{1 \leq j \leq m} a_j} ((l'_1, \ldots, l'_n), (\mu'_1, \ldots, \mu'_n)),$
   where the following are true:

   - $l'_k = l_k, \mu'_k = \mu_k$ for each $k \neq i_1, \ldots, i_m$
   - $(\forall j \in 1..m\ \forall e \in a_j)\ (\exists j' \in 1..m, j' \neq j)\ e \in a_{j'}$
   - $(\forall j \in 1..m\ \forall e \in a_j)\ (\neg \exists i_0 \in (\{1, \ldots, n\} \setminus \{i_1, \ldots, i_m\}))\ (l_{i_0}, \mu_{i_0}) \xrightarrow{a_0} \wedge\ e \in a_0$

Example 3.1. *We illustrate the definitions of timed automata by a variation on an example commonly found in the literature, shown in Fig. 3.4. It consists of a network of two TAs, one modelling a lamp, the other one a person. Each automaton uses one clock, called X for the lamp and Y for the person. We suppose a composition by binary synchronization using the labels "!Push" and "?Push", representing a light switch button being pressed.*

The lamp has three levels of brightness, modelled by the three locations Off*,* Low*, and* Bright*, where* Off *is the initial location (indicated in Fig. 3.4 by the double ring). When the lamp is off (location* Off*), pushing the button switches it on with low brightness (location* Low*). If the next push happens within less than 5 time units then the lamp gets brighter (location* Bright*). If it happens after 5 time units then the lamp is switched off.*

The person has three different activities, each of which corresponds to a location: Dozing*,* Working*, and* Phoning*, where* Dozing *is the initial location. When the person is dozing, he may either push the button once and then phone for at least 1,000 time units or he may push the button twice fast (modelled by an intermediate location) and work for at least 1,000 time units. When he is finished phoning or working, he pushes the button once and gets back dozing.*

Tool implementations

Several tools that perform simulation and/or formal verification on timed automata have been implemented since the 1990s, each with its own dialect. We briefly present some of them in the following and describe how they extend the TA model of Definition 3.8:

- Uppaal [89] extends the TA model with data handling: It is possibile to define local and global variables of simple types (integers and booleans) and structured

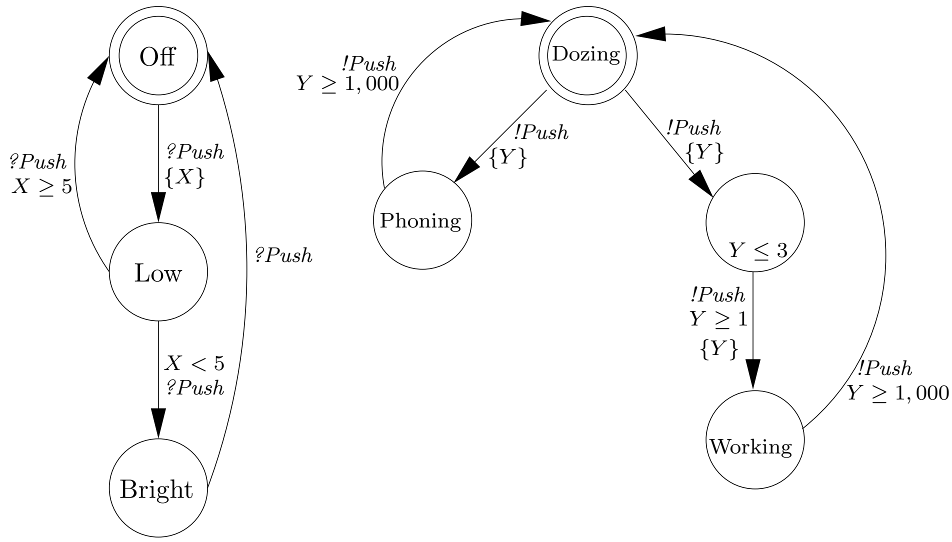

Figure 3.4: Two timed automata describing a lamp (left) and a person (right)

types (arrays and records). Variables can be manipulated by transitions and/or be used in the guard expression of transitions. Uppaal also introduces several time-blocking constructs, such as *urgent transitions*, *urgent locations*, and *commited locations*. Details of the syntax and semantics definition of this TA dialect are given in Section 6.1.1 on page 130.

Uppaal is designed for simulation and Ctl model checking.

- Red [123] also adds local and global variables, but only of simple types (integers and booleans). Communication is implemented using the binary synchronization approach. Red performs Tctl model checking and bisimulation checking.

- Kronos [126] implements an important extension of clock resets during a transition: A clock can either be reset or it can be assigned the value of another clock, which sometimes allows smaller descriptions e.g., the FDDI benchmark [51]. In one automaton, several initial locations may be defined, from which one is chosen nondeterministically. Kronos uses the maximal event synchronization and is designed for Tctl model checking.

- Rabbit [26] encapsulates one or several timed automata in a *module*, from which a hierarchical structure can be constructed. Communication is implemented using the binary synchronization approach.

- Sgm [80] extends the transitions of timed automata with priorities and with different kinds of urgency. Also, the definition of integer variables is possible. Communication is implemented using the binary synchronization approach. Sgm allows model checking on Tctl formulas.

- The tool HyTech [74] applies to *hybrid automata* [73]. Although defined independently, hybrid automata (HA) can be seen as a generalization of timed automata, where clocks are replaced by *hybrid variables*, which can change their value arbitrarily (but possibly within the limits of certain constraints) during the elapsing of time. Intuitively, hybrid variables can be used to represent "environmental" values, such as temperature, pressure, etc. Note that clocks are then special cases of hybrid variables, their value increasing *linearly* at the same rate as time.

Verification is possible, but only with strong restrictions on how hybrid variables evolve [74, 8]; in HyTech, it is limited to reachability analysis.

- Cmc [88] applies to networks of timed automata using general discrete synchronization extended with renaming functions. It performs model checking on properties expressed in the timed modal logic L_ν [87].

  A variation of the tool, named Hcmc [44], applies to hybrid automata.

It should be noted that most of these tools are prototypes used to experiment model checking algorithms rather than robust and mature software tools. The Uppaal tool with its detailed documentation and its graphical user interface is the most important exception to this observation.

3.2.3 Time Petri nets

Formal definition

Different extensions of the Petri net model are the basis for several tools for simulation and formal verification e.g., Tina [16], Roméo [65], CPN Tools [107], Oris [113], and others. We begin by presenting the most influential extension of the Petri net model, initially defined by Merlin [95]. Its central idea is to associate a time interval with each transition, expressing how much time can elapse before this transition is fired. Formally:

Definition 3.9. *Let $\mathbb{I}$ be the set of intervals in $(\mathbb{R}_{\geq 0} \cup \{\infty\})$. A* time Petri net *(TPN) is a 7-tuple $(P, T, in, out, m_0, lab, I_s)$, satisfying the following:*

- *(P, T, in, out, m_0) is a standard Petri net i.e., P is a finite set of* places*, T is a finite set of* transitions*, in, out are mappings from T to multi-sets in P indicating the* in-places *and the* out-places *of a transition, and m_0 is a multi-set in P, called the* initial marking*.*
- *$lab : T \to A$ maps each transition to a label.*
- *I_s is a mapping from T to $\mathbb{I}$. For a transition $r \in T$, $I_s(r)$ is called the* static firing interval.

A TPN is *1-bounded* if at any time, each place contains at most one token. Consequently, such a net contains no transition that has two or more times the same in-place or the same out-place. In this thesis, almost all TPNs we discuss will be *1-bounded* nets. Given this restriction, we will use the following (commonly used) conventions for the graphical representation of TPNs:

- A circle represents a place.
- A dot in a circle represents a token in a place.
- A rectangular box represents a transition. Labels and static firing intervals will occur within or right of the box. Firing intervals that are unbounded for their maximum are written $[x, \infty[$ or $]x, \infty[$.
- An arc (i.e., a directed edge) from a place to a transition represents the place being an in-place of the transition. An arc from a transition to a place represents the place being an out-place of the transition.

Semantics. We now define the semantics of a TPN $\mathcal{R} = (P, T, \mathit{in}, \mathit{out}, m_0, \mathit{lab}, I_s)$, where we will use the following terms:

- A *marking* is a multi-set in P, where each element represents one token in the corresponding place.
- A transition $r \in T$ is *enabled* by a marking m if $\mathit{in}(r) \subseteq m$.

The semantics of $\mathcal{R}$ is defined by a TLTS of the form $(\Sigma, A, \mathbb{R}_{\geq 0}, \rightarrow, (m_0, I_0))$, where $\Sigma, \rightarrow$ are the smallest sets satisfying:

- The states of Σ are tuples of the form (m, I), where m is a marking and I is a partial function from P to $\mathbb{I}$. The domain of I is given by the set of *enabled* transitions.
- $(m_0, I_0) \in \Sigma$, where I_0 is the restriction of I_s to the transitions enabled by m_0.
- *Timed transition*: If $(m, I) \in \Sigma, t \in \mathbb{R}_{\geq 0}$, and for each $r \in \mathit{dom}(I)$, t is in or below $I(r)$, then $(m, \mathit{shift}(I, -t)) \in \Sigma$ and $(m, I) \xrightarrow{t} (m, \mathit{shift}(I, -t))$.

  The auxiliary function $\mathit{shift} : \mathbb{I} \times \mathbb{R} \rightarrow \mathbb{I}$ is used to shift an interval (closed, half-open, or open) to the right (if the second argument is positive) or to the left (if the second argument is negative), while in the second case the shift is limited by zero. Formally (assuming $\wr_1, \wr_2 \in \{[,]\}$):

  $$\mathit{shift}(\wr_1 x, y \wr_2, z) \stackrel{\text{def}}{=} \begin{cases} \wr_1 x + z, y + z \wr_2 & \text{if } x + z \geq 0 \\ [0, y + z \wr_2 & \text{if } x + z < 0 \text{ and } y + z > 0 \\ [0, 0] & \text{otherwise} \end{cases}$$

- *Discrete transition*: If $(m, I) \in \Sigma$, $r \in T$ enabled by m and $0 \in I(r)$, then $(m', I') \in \Sigma$ and $(m, I) \xrightarrow{\mathit{lab}(r)} (m', I')$, where $m' = (m \setminus \mathit{in}(r)) \cup \mathit{out}(r)$ and I' is defined by $I'(r') = I(r')$ if $r' \in \mathit{dom}(I) \setminus \{r\}$, otherwise $I'(r') = I_s(r')$.

  We say that r is *fired*.

Example. We illustrate the definition of time Petri nets by the example shown in Fig. 3.5, describing a bus shuttle between stop A and stop B, and a passenger who wants to take a bus from B to A. Initially, the token in the place "*at home*" indicates that the passenger is at home (next to stop B), and the token in the place "A" indicates that the bus is at stop A. At any moment, the passenger may go to stop B (token in the place "*B (passenger)*"), where she waits five to ten minutes at maximum, before going back home. If a bus arrives in that time, she may board it within one minute, and arrives at A 15 to 18 minutes later. The bus takes 10 to 12 minutes to go from A to B and waits up to one minute before leaving B, either with (place "*passenger in bus*") or without (place "*empty bus*") the passenger.

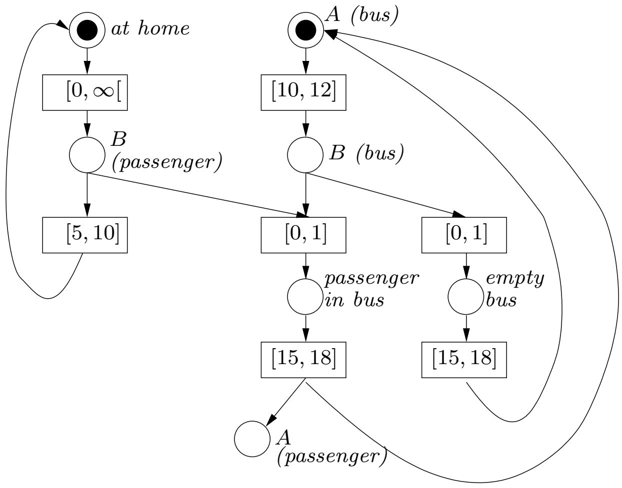

Figure 3.5: A time Petri net describing a bus and a passenger

It is possible to extend this example e.g., by initializing the net with several tokens in the place *at home*. This corresponds to several passengers taking the bus independently.

Variations, extensions, and tool implementations

Several different approaches to extending (time) Petri nets exist:

- In [106], durations instead of time intervals are associated with transitions. Thus, time elapsing occurs as a consequence of the firing of a discrete transition, whereas in the TPN model time elapsing and discrete transitions are independent.

  Such Petri nets with durations are called *timed* Petri nets, and this model is clearly also intuitive. However, timed Petri nets are not compatible with the TLTS model, and it seems difficult to introduce a notion of synchronization. Moreover, very few theoretical results on verification exist, in contrast to the TPN model. Therefore, we will not detail the timed Petri net approach.

- In [36], the authors propose to generalize the definition of TPN to three variations: A *T-TPN* is a TPN as given in Definition 3.9, a *P-TPN* associates time intervals to places (indicating at which time a token created in this place may be used to fire a transition) instead of transitions, and an *A-TPN* associates time intervals to those arcs representing in-places (indicating at which time a token in this in-place may be used to fire this transition)[6].

- There are two main possibilities for introducing priorities into a TPN: Either as a partial order on the set T i.e., a transition may not be fired if a transition with higher priority is enabled [109], or as a partial order on the set of labels i.e., a transition with label l may not be fired if a transition with label l' is enabled and l' has higher priority than l.

  The first approach could for instance be applied in the bus example of Fig. 3.5: The transition representing the bus leaving B *with* the passenger would have priority over the transition representing the bus leaving B *without* the passenger i.e., the passenger takes the bus when it is available.

- In [2], transitions are defined with *inhibitor places* in addition to their in- and out-places. When there is a token in such an inhibitor place, the transition cannot be fired.

  Complementary to inhibitor places are *test places*: A transition with such a test place can only be fired if the latter contains a token; firing the transitions, however, does not consume the token.

It would be possible to extend this list. For instance in recent years, research efforts have been made in defining and analysing TPN extensions such as *Stopwatch*-TPNs, *Scheduling*-TPNs, *Preemptive*-TPNs, etc. [18, 65, 42], but these extensions are not of direct interest in the context of this thesis, so are not discussed.

Several tools for verification on time Petri nets have been implemented since the 1980s. The following list presents briefly some of them:

- The toolbox Tina [16] applies to TPNs extended with priorities on transitions, test places, and inhibitor places. Moreover, it allows variable manipulation by external C functions.

  For verification, Tina provides model checking by an extension of Ltl [22]. It also provides translations to several other model checking tools such as Mec [7] and Cadp [63]. Details on the syntax and semantics definitions of the Tina TPN dialect will be given in Section 6.2.1.

- The tool Roméo [65] applies to TPNs extended with test places, inhibitor places, and stopwatches, and enables model checking either directly of Tctl [3] formulas or indirectly by translations to timed automata.

[6] Unless if we want to point out this difference, we will continue to write "TPN" when we mean "T-TPN".

- The tool ORIS [113] applies to TPNs extended with preemption and probabilistic behaviour. It mainly targets scheduling problems, and enables model checking on formulas of the temporal logic RTTL [102].

- Several tools, such as CPN Tools [107] or ITCPN [1], perform verification on timed extensions of the *coloured Petri net* model. In such a net, a data type is associated with each place and a value ("colour") of this type with each token created in this place.

- The tool INA [117] applies to a discrete time variant of the TPN model, extended with coloured tokens. It provides model checking on CTL [46] formulas.

3.2.4 Other models

Several other models have been defined that are situated at a similar abstraction level to timed automata and time Petri nets, and which also have formal semantics based on the TLTS model. In most cases these models are linked to one specific tool implementation. In the following paragraphs, we give a brief overview of some models (note that it should not be assumed that the corresponding tools are all still maintained).

TASM. The TASM (*Timed Abstract State Machines*) [104] model is also based on the idea of several automata (called *machines*) composed of states and transitions between states.

Real-time behaviour is implemented by associating a duration with each transition, either as a constant or as an interval. Concurrency is implemented by having several machines in parallel. Between these machines, communication is possible by binary synchronization (cf. Section 3.2.2). Data handling is implemented by either global or local variables, whereas data transmission by synchronization is not possible.

The main objective of the TASM language is the representation of resource consumption e.g., to enable the calculation of the *worst or best case execution time*. The latter is implemented by a translation of a subset of TASM to UPPAAL timed automata.

Verus. The Verus [43] model handles processes whose description is strongly inspired from programming languages. In terms of automata-based languages, a Verus process has a single state with a single transition, where the transition is expressed using a complex syntax that combines (deterministic and non-deterministic) assignments (to boolean and integer variables) and delay statements by conditional structures and loops.

Real-time behaviour is implemented in a simple and rigid way by constant delay statements, using a discrete time domain. Concurrent processes can only communicate by global variables.

Formal verification is defined on formulas of the temporal logic RTCTL [55], which is a real-time dialect of CTL.

Clocked Transition Modules. The CTM [28] model, the input language for the tool STeP, can be seen as a variation of networks of timed automata, in which no locations are defined. Instead, each process (called a *clocked transition system*, CTS) is defined by several transitions that can only be executed when a constraint formula on the variables (discrete variables and clocks) is satisfied. These transitions are labelled and may contain variable assignments.

The time domain is dense. Parallel composition between different CTSs is implemented by a simple form of the maximal event synchronization (cf. Section 3.2.2), where each label corresponds to a single event. Furthermore, communication is possible by global variables.

I/O interval structures. I/O interval structures are the input model used by the tool Raven [110]. In this model, each such structure is a very simple variant of a timed automaton: The time domain is discrete, and each transition has a time constraint that depends on the time elapsed since the last discrete transition. Time can only elapse if a discrete transition is possible in the future (this is a rigid version of the "invariant" construct from the TA model).

Communication between different I/O interval structures is only possible by boolean variables; there is no synchronization.

The tool implements formal verification by the temporal logic CCTL (which is an extension of CTL).

3.3 High-level languages

A graphical approach is rarely appropriate for modelling of realistic systems, for several reasons. For example, in a graphical approach, the set of processes is static (no processes can be created or terminated during execution), communication is limited to simple concepts (binary or global synchronization, cf. Section 3.2.2), and the structure of states and transitions is rigid and can be artificial. In short, the simplicity of the provided constructs can make the modelling of complex systems a cumbersome task.

To palliate this, *high-level modelling languages* have been developed. Those languages are purely textual and provide powerful constructs that enable modularity, composability, and concise notations. In particular, they avoid a state/transition-structure, with the benefit of a relatively free composition of statements for communication, along with control structures that allow different kinds of composition, such as sequential composition, parallel composition, and repetition by recursion or by loop constructs.

Not all formal description techniques provide formal semantics, as pointed out in [40]. However, a formal semantics definition is clearly necessary to perform formal verification. Therefore, we only consider high-level languages defined with a formal semantics.

Among these, *process algebras* [14] are a natural choice: The semantics of a process algebra is given by a set of rules, each one transforming one syntax term into another syntax term, where each transformation corresponds to a transition in a (timed) labelled transition system i.e., syntax terms also correspond to semantic states. These rules are usually defined in the simple style of Plotkin's structural operational semantics [105].

Process algebras express the communication between concurrent processes in form of a mutual synchronization, as opposed to buffer communication (where sending and receiving of a message are two successive events) or shared variables (which opposes the intuition of independent systems). Such a synchronization is considered to be performed simultaneously in two or more processes on a *gate* (or communicating port), which generalizes the idea of labels as they appear in semantic and graphical models.

The expressive power of the first process algebras (around 1980) included control structures such as sequential and parallel composition of terms, choice between terms, and recursion. During the 1980s, extensions introduced data representation and manipulation, and finally real-time concepts were implemented in e.g. [108].

There are also well-known problems shared by all high-level languages, such as the steep learning curve encountered by system designers (caused by the high level of abstraction) that still hinders broad industrial application. Therefore, some effort has been put on defining languages considered as "*next generation languages*", such as E-Lotos [83] (which became an Iso standard in 2001) and Lotos NT [115], that combine the strong theoretical foundations of process algebras with language features suitable for a wider industrial dissemination of formal methods. Such features include complex data structures, a more flexible parallel composition, and exception handling.

In the remainder of this section, we will give short descriptions of those languages that were, directly or indirectly, of influence for the definition of Atlantif.

3.3.1 Languages based on CCS

The seminal language CCS (*Calculus of Communicating Systems*) proposed by Milner [97] is mainly aimed at studying theoretical problems. The syntax of CCS composes communication actions[7] by constructs expressing e.g., concurrency, action prefixing, and choice. Due to the simplicity of its syntax and semantics, CCS influenced most other high-level languages, at least in parts. However, CCS does not include data and real-time aspects.

The first real-time extensions appeared around the year 1990. We only cite two of them: *Temporal CCS* [98] introduces operators for either fixed or unspecified time delays, which can be used to derive more complex constructs, such as *time-outs* (cf. 1.1.1). The time domain is considered to be discrete.

Timed CCS [125] also provides fixed delays, but extends the unspecified delays by introducing a *time capture* operator "@": The construct "$\mu@t.P$" may delay a certain time, then perform the action μ; afterwards, each occurrence of t in P (e.g., as parameter for delays) is replaced by the delay elapsed before performing μ. Moreover, the time domain is generalized to be either discrete or dense.

[7] [97] states that "the behaviour of a system is exactly what is observable, and to observe a system is exactly to communicate with it".

3.3.2 Languages based on CSP

The language CSP (*Communicating Sequential Processes*) was developed by Hoare [77] around the same time as CCS, but with a stronger orientation towards industrial applications by a more pragmatic approach. CSP does not include data and real-time aspects.

TCSP (*Timed CSP*) [108] extends CSP with an independent fixed delay action. It also supposes a constant delay value that always elapses between two discrete transitions, thus avoiding strict zeno behaviours (cf. Definition 3.6). Moreover, Timed CSP supposes an orthogonal time concept i.e., elapsing of time has no influence on which actions can be performed. The time domain used is the set of positive real numbers.

TCSP was revised in [103], most notably by deleting the constant delay between two discrete transitions; it also introduces a concept of "signals". As in [108], this revision does not introduce data-related constructs.

In [58], the language CSP-OZ is defined as a combination of CSP with Object-Z [53]. The Object-Z part provides data in form of object-oriented type definitions. In [78], CSP-OZ-DC is defined as a futher extension of CSP-OZ with a subset of Duration Calculus [127], providing real-time constructs.

3.3.3 Languages based on LOTOS

The language LOTOS [81] (*Language Of Temporal Ordering Specification*) is an ISO standard for formally describing concurrent and communicating systems. It also provides data types, which can be either predefined or user-defined algebraic data types. Such type declarations are strictly separated from the control part of a LOTOS specification. The latter is defined in the usual process algebra style, inspired from both CCS and CSP.

Like for CCS and CSP, many extensions of LOTOS have been proposed, of which we only cite the most relevant to this work. Regarding real time, the first extensions occurred around 1990, e.g.,T-LOTOS [31] and perhaps most influential, ET-LOTOS (*Enhanced Timed* LOTOS) [91]. ET-LOTOS reuses the idea of a time capture operator as defined in Timed CCS (cf. Section 3.3.1), and at the same time, it extends this construct with a second meaning: An action with an attached "@t" may optionally have also attached a boolean formula SP, which expresses how much time may elapse before the action occurs i.e., if d time units elapse, the action can only occur if $[d/t]SP$ evaluates to **true**. Furthermore, ET-LOTOS provides a fixed delay operator and a generalized (i.e., either discrete or dense) time domain.

RT-LOTOS (*Real Time* LOTOS) [47] is based on the ideas of ET-LOTOS, but contains several differences. First, the powerful but sometimes cumbersome time capture is replaced by a much simpler (and less expressive) constant *time restriction* of the form "$\{d\}$" (corresponding in ET-LOTOS to a formula $SP = t \leq d$). Second, RT-LOTOS introduces a *latency* operator, which can delay urgent actions. Another further extension to ET-LOTOS is defined in [76], where a "suspend/resume" mechanism is proposed.

3.3.4 E-LOTOS and LOTOS NT

Different ideas on how to extend LOTOS converged into E-LOTOS (*extended* LOTOS) [83], also an ISO standard. The main objectives for this revision were to enhance the user friendliness (e.g. with data definitions closer to programming languages), but also to enhance the expressive power. The extensions include a generalized parallel composition (a single construct that expresses for several gates and several processes the combination of processes that can synchronize on which gates), symmetrical sequential composition (both the left and the right side of a sequential composition can be arbitrary terms, whereas the prefixed sequential composition of LOTOS only permitted atomic actions on the left side), exception handling (unexpected events may raise an *exception*, which leads to the execution of a specially defined code, the *exception handler*), and of course real-time constructs. The latter are mostly inspired from the ideas introduced in ET-LOTOS.

The language LOTOS NT (LOTOS *Nouvelle Technologie*) [115], which has been developed in parallel with E-LOTOS, can be seen as a dialect with small differences in syntax and semantics.

3.3.5 Other high-level languages

The language ACP (*Algebra of Communicating Processes*) [13] features in particular a *communication function*, which defines the synchronization of concurrent processes independently of the process terms, thus enabling a simpler notation of processes. Neither data nor real-time constructs are provided.

Different timed extensions for ACP are defined and discussed in [85]. The ideas for the additional constructs are similar to those discussed in the preceding sections i.e., fixed delays, time restrictions, and urgency are provided, and the time domain is discrete. As we already mentioned in Section 3.1.2, the semantics are defined by transition systems providing labels with actions and absolute time stamp, instead of the TLTS model.

The language μCRL (*micro Common Representation Language*) [71] extends ACP with abstract data types. In [29], a real-time extension is proposed, where the time domain is discrete and time elapsing is represented by a special "tick" action, on which all processes have to synchronize. The revised version mCRL2 [70] extends μCRL regarding data representation and communication.

Another early timed language is ATP [100], which extends a process algebra (with similarities to CCS and ACP) with a fixed delay (of one time unit), urgent actions, and time restrictions for actions. The originally discrete time domain is later generalized into an arbitrary time domain. Data types are not provided.

We finish this necessarily highly incomplete list by mentioning Timed χ [120], which has the interesting characteristic of not being the extended but the simplified version of another language: Timed χ restricts the hybrid variables provided by Hybrid χ [121] to timed variables (cf. Section 3.2.2 on page 28), which enables the expression of delayable and urgent actions, independent fixed delays, and a dense time domain.

3.4 Intermediate models

Inconveniences in the approaches of graphical models and high-level languages have been observed for several years, in particular the following:

- The development of verification tools for specifications given in high-level languages providing concurrency, data, and time is difficult: The TLTS defined by such a specification is usually infinite, even in a simple case (cf. Section 3.1.1 on page 21). Thus, symbolic representation techniques would have to be found for the tool. Moreover, high-level languages have complex rules where a transition often depends on the existence of other transitions (e.g., by urgency constraints as described in Definition 3.43.), which is not trivial to check in an infinite model.

  In practice, verification tools for such high-level languages only apply to significantly limited subsets of the languages [94]. Therefore, although high-level languages are appropriate for modelling complex systems, their practical application for verification purposes is restricted.

- Although graphical models basically share the problems of infinite semantic models, their syntax and their semantics rules are much simpler, which makes the development of verification tools posssible.

  But at the same time, as discussed in the introduction of Section 3.3, modelling of complex systems using graphical models is often difficult.

Thus, there is clearly a gap between the modelling in high-level languages and the verification in tools conceived for graphical models. Several works [37, 61, 21] proposed to fill this gap by *intermediate models* that satisfy the following:

- They have constructs of high-level languages, such as user-defined data types, choice operators, complex communication definitions, or action-independent delays.
- They enable formal verification either indirectly (i.e., via a translation to another, ususally graphical, model) or directly. Providing this possibility usually means that the model has similarities to graphical models.

In the sequel, we briefly present the central ideas and the most distinctive features of a few intermediate models. Complete definitions of their syntax and semantics are beyond the scope of this work.

3.4.1 IF and IF-2.0

The IF (*Intermediate Format*) model [37] extends networks of automata. A specification consists of several processes that can communicate by sending and receiving *signals* and by dynamically creating and deleting other processes. A communication by a signal corresponds to two different discrete transitions: one for the emission and one for the reception.

The approach is therefore different from the communication by synchronization used in most other models discussed in this chapter. The IF model supports data, providing several predefined types (booleans, integers, and floats), user-defined types (constructed by enumerations, records, and arrays), and the possibility to import externally-defined data types.

Each process contains several discrete states. Transitions between discrete states are defined by a precondition (i.e., a condition that has to be satisfied *before* the transition is taken) and an action (i.e., variable assignment, signal emission, signal reception, starting of a process instance, or stopping of a process instance).

The revised version IF-2.0 [38] introduces constructs for real-time behaviour in a very similar way to timed automata i.e., using clocks (cf. Definition 3.7) that can be used in preconditions and that can be reset in actions. Furthermore, a transition between discrete states also contains a tag that expresses whether the transition is urgent. The time domain is dense.

The formal semantics are defined using two relations: first a relation defining the discrete transitions in a single process, and a second relation, defined using the first relation, for discrete transitions of one process among many and for time elapsing in all processes.

The IF model is conceived as a pivot language in translation chains from semi-formal models, such as UML and SDL, to different tools such as KRONOS (cf. Section 3.2.2) and toolboxes such as CADP [63] to perform verification and simulation. Therefore, the syntax and semantics of IF are designed to meet models as UML and SDL.

3.4.2 BIP

The BIP (*Behaviour, Interaction, Priority*) [11] model also supposes several concurrent processes (called *components*). The central idea is to define specifications with a clear separation between three different levels [67]: First, behaviour of single components, second, interaction between those components, and third, priorities that control the interactions. Data can be defined in the form of local variables, where the available types are those of the C programming language.

Each component contains several discrete states. Transitions between discrete states are defined by a precondition, a label (called a *port*), and a variable update.

Communications between processes are defined using *connectors*, which can have the form of either a rendezvous or of a broadcast. A rendezvous connector enables one, two, or several components with a certain port each to synchronize. A broadcast connector enables a single component (the emitter) with a certain port and an arbitrary set of other components with certain ports (the receivers) to synchronize. A connector can also assign local variables in the synchronizing components, possibly parameterized by local variables of another synchronizing component.

BIP provides a modular structure: Instances of components can be grouped into *compounds*, which themselves may also be part of another compound.

Real-time behaviour can be defined for single components. Transitions in such components may be defined as urgent. If no urgent transition is enabled, time may elapse (only on a discrete domain). Hybrid variables can be defined in a timed component and therefore change their values during the elapsing of time.

3.4.3 AltaRica

The *Timed* AltaRica [45] model shares many properties with If-2.0 and Bip: It is also based on a set of concurrent processes (called *components*) with local variables. Simple data types (booleans, integers, and enumerations) are provided, as well as user-defined types (using records and arrays). Components can be grouped (into *nodes*) to achieve modularity. Synchronizations in AltaRica correspond to the *rendezvous* communications of Bip. Priorities can be defined between transitions.

The real-time behaviour is implemented in a way similar to timed automata, using clocks, clock constraints on transitions, and invariants on discrete states. The time domain is dense.

In [45], a translation from Timed AltaRica to timed automata in the Uppaal dialect is defined, thus enabling verification.

3.4.4 MoDeST

The MoDeST (*Modelling and Description Language for Stochastic Timed systems*) [30] is also based on a set of concurrent processes. These sequential processes are defined in the style of high-level languages i.e., atomic actions such as communications and variable assignments are composed by operators such as sequential composition and choice. With this structure, MoDeST is the only intermediate model presented here that is defined without discrete states.

Moreover, MoDeST stands out as having several features not provided in the other intermediate models: The choice between different actions can be extended to a probabilistic choice, exception handling can be used, and variables can be assigned nondeterministically. Real-time syntax is defined by clocks, clock constraints on actions, and a concept of urgency that is similar to the invariant construct of timed automata. Communication between parallel processes is provided by synchronization on gates (for each gate, a synchronization is defined among all processes using this gate) and by shared (global) variables.

The semantic model defined for MoDeST is a probablistic extension of the timed automata model.

3.4.5 Promela

The Promela (*PROcess MEta LAnguage*) [79] model is the input language for the model checking tool Spin, which checks formulas of the temporal logic Ltl. The model generalizes the automata model, by defining *processes* which can be instantiated instead of rigid automata.

Although based on the graphical intuition of states and transitions between states, the syntax of Promela is purely textual and similar to programming languages like C. The standard syntax of Promela does not provide constructs that express real-time aspects. The Promela dialect defined in [35] provides an extension with real-time constructs, where the time domain used in this model is discrete.

3.4.6 NTIF

The Ntif (*New Technology Intermediate Form*) [61] model has been conceived to represent sequential processes handling complex data structures. Ntif has no constructs expressing concurrency or real time, but those were intended for future work (and this thesis should be considered as a part of such future work).

An Ntif process is an automaton containing a set of control states. Each state is associated with a statement called a *multibranch* transition. Such transitions are defined using high-level standard control structures (such as deterministic and nondeterministic variable assignments, **if-then-else** and **case** conditionals, nondeterministic choice, and **while** loops) and communication events. This approach enables a representation of processes that is more compact than in models using *condition/action* transitions (i.e., transitions defined by a precondition and a communication action or an assignment).

A translation from Ntif to the If model (cf. Section 3.4.1) has been defined. More recently, Ntif has found industrial applications in the framework of the Topcased[8] project led by Airbus.

3.4.7 Fiacre

The Fiacre (*Format Intermédiaire pour les Architectures de Composants Répartis Embarqués*) [17, 15] model is mainly based on the two models Ntif and V-Cotre. V-Cotre [21] is also an intermediate model defined to compile higher-level specifications into timed automata, time Petri nets, and transition systems.

A Fiacre process provides roughly the same constructs as an Ntif automaton, in particular the concept of discrete states associated with multibranch transitions. Several concurrent processes can be composed in a hierarchical structure (inherited from V-Cotre) and communicate using synchronization vectors [6, 33], which enables synchronizations on different gates among one, two, or several processes.

[8] http://www.topcased.org

It is also in these synchronization vectors that real-time behaviour is implemented: Each vector may define a time interval using an approach very similar to that of time Petri nets (cf. Section 3.2.3), notably, reaching a limit of such an interval blocks the elapsing of time. Note that timed constraints cannot be specified explicitely in sequential processes.

Fiacre is situated as a pivot element in translation chains. Translations into Fiacre have been defined from the industrial models Aadl and Sdl, and translations from Fiacre have been defined into the language Lotos (cf. Section 3.3.3), which creates a connection to the Cadp [63] toolbox, and into the time Petri net dialect of the tool Tina [23] (cf. Section 3.2.3).

Remark 3.1. *It should be noted that a separation line between graphical and intermediate models is not easy to draw; it is clearly debatable whether some models listed here would be better in Section 3.2.4 or vice versa. As a rule of thumb, an intermediate model corresponds to a graphical model that firstly is extended significantly with language features borrowed from high-level languages and secondly provides a translation to one or several tools based on graphical models for verification purposes. Detailed descriptions of such approaches are given e.g., in [37, 61].*

3.5 Summary and observations

3.5.1 Possible approaches: a summary

In this section, we give an overview of the choices that can be made when defining a language or a model expressing data, concurrency, and real time, based on the analysis of the models presented in this chapter. This overview will be the basis for the design choices that we will make in the next chapter.

Moreover, it will allow us to understand more precisely what exactly is meant by data, by concurrency, and by real time, which we only described on a more intuitive level before (cf. Section 1.1.2). It can be seen that design choices may have an impact on the syntax, on the semantics, or on both (in various relations).

Semantic Model

As we said above (cf. Section 3.1.1), we see the Tlts model as a common basis for the more abstract models and languages presented in this chapter. Indeed, timed automata, time Petri nets, and most of the other models define their semantics using Tltss. We can conclude that one important assumption (described in the same section) of the Tlts model is broadly agreed to: Discrete actions take no time.

Nevertheless, not all languages and models make this assumption, and they are thus defined with other semantic models e.g., the different extensions of ACP mentioned in Section 3.3.5. Moreover, I/O interval structures (cf. Section 3.2.4) suppose a duration of one time unit for each communication action.

Data

Some of the models and languages described in this chapter do not handle data. For the others, these are two of the most important differences in the approaches:

- **Data types:** Most models providing data also provide user-defined types. The modelling of complex systems in a simple and concise way often requires grouping of data in constructs such as arrays and records. This can be illustrated by simple examples such as those given in Section 1.1.2.
- **Operators and functions:** All models with data handling provide basic operators (like addition and multiplication) for the manipulation of variables. Some models additionally provide the possibility of user-defined functions, either integrated in the model's syntax (e.g., in Uppaal timed automata, cf. Section 3.2.2) or imported from another language such as C (e.g., in Tina time Petri nets, cf. Section 3.2.3).

Concurrency

Almost all models and languages described in this chapter provide concurrency between processes. We list below some of the most important differences in the approaches used:

- **Buffers and synchronizations:** Most models provide the possibility to communicate via gates, in one of two different ways: First, by *synchronization* i.e., the communication happens at the same time in all participating processes. Second, by *buffers* i.e., one process sends, and afterwards, another one receives. The latter approach is followed by the If-2.0 model (cf. Section 3.4.1). All the other models with gate communication use synchronization.

  Note that buffer communication is strictly less expressive than synchronization, because a buffer can be expressed in a model that communicates by synchronization, but not the other way around.
- **Synchronization notation:** For models and languages that use synchronization, two approaches to represent synchronizations are mentioned in this chapter:

  First, synchronizations can be described directly i.e., each possible occurrence of a synchronization is represented by its own construct. This is the approach of Petri nets, thus used in the TPN model (cf. Section 3.2.3).

  Second, synchronizations can be described indirectly i.e., the processes are defined with occurrences of gates, and how synchronizations between these gates is possible is defined by another construct (which will be the subject of the next point below). This approach is used by all synchronizing models and languages except the TPN model.
- **Synchronization syntax:** For models and languages that use an indirect synchronization notation, such notations have different expressive power. The simplest

forms are the binary synchronization (synchronizing two processes) and the general discrete synchronization (synchronizing all processes), already described in the context of timed automata (cf. Section 3.2.2). Binary synchronization is used not only in different dialects of the TA model, but also e.g., in the PROMELA and TASM models (cf. Sections 3.4.5 and 3.2.4).

More complex forms of synchronization, which allow an arbitrary number of synchronizing processes, exist in many different variations e.g., the connectors we already described for the BIP model (cf. Section 3.4.2).

- **Separation of sequential and parallel behaviour:** All intermediate models presented in Section 3.4 define sequential processes as rigid structures that can be put in parallel to enable concurrency. In contrast, process algebras are more general since they allow to freely combine sequential and parallel operators. This feature, clearly useful for concise modelling, can be translated into graphical models for the untimed case [59, 84]. In combination with real time however, a translation preserving the semantics becomes a much more complex problem [115, 112].

- **Starting and stopping:** Some of the models presented provide constructs to start and stop processes i.e., the set of processes that can interact may change dynamically. For instance, this is possible in PROMELA (cf. Section 3.4.5) and in IF-2.0 (cf. Section 3.4.1).

- **Data transmission:** In languages and models providing concurrency and data handling, two approaches are followed for transmitting data values between concurrent processes.

  First, data can be transmitted using *global variables*, which are written by one process and read by another process. This approach is used in most graphical models e.g., the UPPAAL dialect of timed automata (cf. Section 3.2.2).

  Second, data can be transmitted using gates extended with *offers* e.g., instead of simply synchronizing on a gate G, we also exchange the values "5" and "**true**" at the same time. LOTOS and its variations (cf. Section 3.3.3) use this approach. A variation of the offer approach can be found in the BIP model, where variable assignments that may read and write variables of all synchronizing processes can be executed during a synchronization.

  Both approaches have inconveniences: Shared global variables can lead to access conflicts, and offers need to be defined by additional language constructs. Note that the FIACRE model even combines both global variables and synchronizations with offers.

Real time

Most models and languages presented in this chapter provide constructs expressing quantitative time. Numerous aspects in the representation of time are treated by different approaches, discussed in the following.

Time elapsing speed, absolute time, and relative time. In most models combining concurrency and real time, time advances in every process with the same speed. Yet, two intermediate models abstract from this idea: In a specification of the Bip model, a subset of the processes may be described with only qualitative time (i.e., the order of events), and in the Fiacre model, all sequential processes are only described with qualitative time, while quantitative time constraints are described at the level of the parallel composition of processes.

It is well-known that the assumption of time progressing everywhere at the same speed is not entirely true in reality (because it contradicts general relativity [54]), but clearly, it is still an intuitive abstraction. Note that hybrid models can easily express specifications where the assumption is not satisfied.

For time that progresses everywhere at the same speed, two basic representation approches can be thought of: Either it can be *absolute* i.e., counting from the beginning of the system's execution; or it can be *relative* i.e., counting from the last action.

For instance, the absolute time description "5 time units after the start do A, 8 time units after the start do B" corresponds to the relative time description "after 5 time units do A, after 3 more time units do B". The vast majority of the models discussed here use relative time. An interesting exception is the timed automata model (cf. Section 3.2.2), which mixes the two approaches: A clock that is reset each time a discrete transition is taken measures relative time, and a clock that is never reset measures absolute time. A detailed discussion of the difference between absolute and relative time can be found in [10].

Occurrence level of timed syntax. There are two levels at which real-time behaviour can be described:

1. Within a process, the occurrence of delays and/or restrictions to discrete actions is possible i.e., each synchronization between several processes is constrained by the intersection of several individual restrictions. This approach, called "timed-action" in [90], is used in timed automata, in all high-level languages extended with real time presented here, and in all the intermediate models described above, except Fiacre.

2. Time constructs are not defined in the individual processes, but in their parallel composition. This approach, called "timed-interaction" in [90], is thus limited to those models that communicate by synchronization. For instance, each static firing interval in the TPN model (more precisely the T-TPN model, cf. Section 3.2.3 and Definition 3.9) is associated with one synchronization. Similar intervals are associated with each synchronization vector of the Fiacre model.

   However, many models and languages that use the timed-action approach also contain constructs for timed-interaction, such as the possibility to **hide** a synchronization in ET-Lotos (cf. Section 3.3.3), which makes it *urgent* and thus, introduces a time restriction on the synchronization level. The *urgency* defined in several dialects of timed automata works similarly.

Depending on what is to be modelled, both approaches have a valid intuition, but using both would obviously make syntax and semantics much more complex to define.

Clocks. A few models and languages use *clocks* (cf. Definition 3.7) e.g., timed automata, clocked transition modules (cf. Section 3.2.4), and AltaRica (cf. Section 3.4.3). As described in the semantics of timed automata (Section 3.2.2), clocks are variables whose value increases by x after a timed transition that lets x time units elapse.

However, shifting values from one state to the next (by a timed transition) can also be identified in all other models and languages that use a dense time domain. For instance, in time Petri nets, the time intervals of a state would be shifted by x (cf. Section 3.2.3, function *shift*), or in real-time process algebras, time-related terms would be rewritten by a reduction of x. Thus, clock models do not really differ in semantics from non-clock models.

Time domain. Concerning the time domain, two fundamental approaches exist: Some real-time models enable a unique time domain (e.g., timed automata, timed AltaRica, or ACP$_\rho$ [9] with a dense domain, and Bip or temporal CCS with a discrete domain). Some other models allow both in different specifications i.e., they do not make a choice between a dense or a discrete domain. For instance, *Timed CCS* [125] and E-Lotos [83] follow this approach.

Differences between a dense and a discrete domain were already discussed in Section 3.1.1.

Life reducers and time capture. Most models provide the concept of a *life reducer* in one form or another i.e., a construct attached to a discrete action, which defines the set of instants at which the action is possible. This set of instants can be defined using different constructs, four of which are described in the following list:

- In the timed automata model, the life reducer takes the form of a guard formula attached to a transition (cf. Definition 3.8).
- In the time Petri net model, the life reducer takes the form of a static firing interval attached to a transition (cf. Definition 3.9).
- In the language RT-Lotos (cf. Section 3.3.3) the life reducer takes the form of a single numeric value attached to a discrete action, and represents how long synchronizing this action is possible. In combination with an independent delay construct, this provides the definition of a time interval, as for time Petri nets, but with two important differences: First, it operates on the process level (cf. the above paragraph on the occurrence level of timed constructs), and second, only *closed* intervals are possible i.e., open or half-open intervals are excluded. Other languages such as ATP (cf. Section 3.3.5) use similar constructs.

- The language ET-Lotos provides discrete actions with a time capture "@t" and a formula "[SP]" as described in Section 3.3.3. Similar to the guard formula of timed automata, these constructs allow more complex constraints than mere intervals e.g., "$t \leq 3 \vee t \geq 5$".

As already described in Section 3.3.3, the time capture of ET-Lotos also has a second function, similar to Timed CCS: The variable V remains assigned with time value after the action, such that it can be used in further assignments, conditions, etc.; thus information is kept that would be lost otherwise. Although such a construct easily expresses behaviour that could not be expressed elegantly, if at all, in a language such as RT-Lotos, experience shows however that it is too powerful in the general case to define algorithms for formal verification. Therefore we consider it of limited benefit.

Behaviour when a life reducer reaches its limit. In general, the set of instants defined by a life reducer may contain a maximum, namely the latest instant at which the communication action may happen. When this instant has been reached, two semantics can be defined:

- Time cannot elapse anymore from the current state.
- Time may elapse, but the communication action can no longer occur in states reachable by timed transitions.

In the literature, different names have been given to these two approaches. We will speak of a *strong deadline* in the first case and a *weak deadline* in the second case, inspired by [36] that speaks of *strong timed semantics* and *weak timed semantics*. [90] speaks of *must timing policy* and *may timing policy*, whereas the expressions of *hard* and *soft real-time properties* are used e.g., in [40].

Several models and languages of this chapter allow constructs for strong deadlines as well as constructs for weak deadlines. E.g., the If-2.0 model controls this by the urgency label of a transition (cf. Section 3.4.1). Other approaches can also be found:

- The Fiacre model and the time Petri net model are restricted to strong deadlines.
- I/O interval structures (cf. Section 3.2.4) [110] implement a compromise between weak and strong deadlines: From each discrete state, an outgoing transition t uses a weak deadline if and only if another outgoing transition is possible after time elapses beyond the time limit of t.
- In [92], it is observed that the semantics of a delay (forcing an amount of time to elapse) is easy to define, whereas forcing an action to happen within a certain amount of time is quite complex. Therefore, ET-Lotos (cf. Section 3.3.3) and other languages only allow strong deadlines for hidden synchronizations i.e., synchronizations that are entirely independent of the environment, while other real-time constructs are restricted to weak deadlines.

The last point shows that urgency is a related topic of strong and weak deadlines. Further discussions can be found in [32, 90].

Additivity, determinism, and maximal progress. Although most languages and models use the TLTS semantic model, it is in general not always guaranteed that the three "good properties" of time additivity, time determinism, and maximal progress given in Definition 3.4 are satisfied, as discussed in the following remark:

Remark 3.2. *The intuitive properties of well-timedness (1, 2, and 3 of Definition 3.4) are often difficult to implement in the formal semantics definition of a language. For instance, the following issues have to be considered:*

- *Most languages contain a construct for expressing a delay. For a delay of t time units, time additivity obliges the semantics to not only define one transition labelled t, but also a pair of additional transitions for each t_1, t_2 such that $t = t_1 + t_2$ (of which there are an infinite number, if the time domain is dense).*

  *Moreover, suppose three states S_1, S_2, S_3 such that $S_1 \rightarrow t_1 S_2 \rightarrow t_2 S_3$. Then, time additivity imposes $S_1 \xrightarrow{t_1+t_2} S_3$ i.e., it obliges the semantics to "know" in S_1 (*before *the first delay) that in S_2 another delay of duration t_2 will be possible. More generally, because of the transitive closure of the timed transition relation, the semantics must consider not only the time elapsing possible in a given state, but also the time elapsing in (timed) successor states of a given state, in successor states of successor states, etc.*

- *Most languages contain a construct for expressing nondeterministic choice. If time elapses after such a choice, time determinism obliges the semantics to represent* each *possible evolution following this choice. This makes state representation difficult.*

  For instance, consider a process that expresses that a variable x is nondeterministically assigned an arbitrary integer as value, followed by a delay of five time units. The state that is reached after these five time units have elapsed must contain information about the value of x. This information could not be "x equals one" or "x equals two", because each of these cases would exclude all other cases ("x equals three", etc.). Thus, an infinite number of possible values for x must be expressed in the state after the time elapsing.

- *Syntax rules respecting the maximal progress of urgent actions must obviously contain a restriction for the elapsing of time, expressing that time must not elapse, if during the elapsing time an urgent transition might occur. Formally, this restriction imposes rules with* negative premises *i.e., a hypothesis of the form "$\neg\exists\varphi$", where φ describes urgent transitions. As discussed in [68, 69], negative premises can cause inconsistency in the semantic rules.*

For instance, RT-LOTOS does not satisfy time additivity, because its rules do not foresee future time elapsing. E.g., the term "$\Delta^2\Delta^2 G\{4\}; \mathbf{stop}|[G]|\Delta^3\Delta^2 G\{0\}; \mathbf{stop}$" (which is a parallel composition of one sequence “wait 2 – wait 2 – synchronize on G within 4 – stop” with another sequence “wait 3 – wait 2 – synchronize on G at once – stop”) can be used to derive a chain of timed transitions labelled 2, 1, 1, and 1, followed by a discrete transition labelled G, but not a timed transition labelled 5 before a discrete transition labelled G.

Time determinism is nearly always satisfied, but sometimes e.g., in the Fiacre model, it comes at the price of introducing additional τ-transitions in the Tlts. In other cases e.g., in the E-Lotos language, it comes at the price of a large number of semantic rules.

The property of maximal progress of urgent actions (cf. Definition 3.4 *(iii)*) is situated on a more technical level than the first two: Its satisfaction basically depends on whether the model is defined with a notion of urgency or not. Most models do have urgency, but it takes different forms. In several process algebras (e.g., Lotos NT), it is assumed that τ-transition (i.e., hidden) equals urgent transition. RT-Lotos does not share this assumption; instead it uses the refinement “hidden transition of which the latency time has elapsed equals urgent transition”.

Other Aspects

It should be remembered that the differences between the languages and models described in this chapter are not restricted to properties related to data, concurrency, and real time. Other aspects such as hybrid variables, probabilities, etc. also appear in different ways in a few models.

We consider these aspects to be out of the scope of this work and will not discuss them further.

3.5.2 Observations from the comparison

The preceding sections showed that there are several fundamental differences in the approaches to how data, concurrency, and real time are implemented in high-level languages and graphical models. Design problems caused by the combination of these three aspects, such as the occurrence level of timed syntax, illustrate the need for rather complex semantic definitions.

Considering the different approaches and the complex rules, it is understandable that approaches defining automated translations directly from high-level languages to graphical models are very complex and raise problems of semantics preservation e.g., [111].

Therefore, it seems reasonable to define, instead, translations between formalisms with smaller differences i.e., to follow the approach of *intermediate models*: High-level syntax constructs (e.g., for data manipulation and synchronization) and high-level semantics in intermediate models enable translation (manually or automatically) from high-level languages to a convenient intermediate model. A structure similar to the automata model used in most intermediate models enables (automatic) translations from intermediate models to a convenient graphical model.

However, it can be observed that existing intermediate models are not sufficient to intuitively represent important concepts present in elaborate high-level languages such as E-Lotos and Lotos NT (cf. Section 3.3.4)[9]:

- The If-2.0 model provides no synchronization.
- The Bip model only provides discrete time.
- The Ntif model is restricted to sequential untimed processes.
- All models except Fiacre and Ntif provide only transitions with a simple condition/action syntax, which does not enable efficient modelling and compilation.
- The Fiacre model uses timed syntax and semantics following different approaches from most high-level languages, namely regarding the occurrence level of timed syntax and the limitation to strong deadlines.

Therefore, it is justified to define a new intermediate model which is suited better for this task. This will be the subject of the following chapter.

[9] The reason for this lack is that the definitions of those intermediate models are intended to represent other languages than E-Lotos or Lotos NT e.g., UML and SDL for If-2.0 or probabilistic languages for MoDeST.

Chapter 4

The syntax and semantics of ATLANTIF

Abstract This chapter defines and discusses the ATLANTIF (*Asynchronous Timed Language Amplifying* NTIF) intermediate format. It begins with a syntax overview along with a brief description of semantics, showing our design choices, and then describes formally each construct, namely its syntax, its static semantics restrictions, and its dynamic semantics. Finally, semantic properties of ATLANTIF are analyzed and discussed.

4.1 Syntax and semantics notation

Syntax metalanguage The syntax definitions of this chapter will use a variant of EBNF (*Extended Backus-Naur Form* [82]). The EBNF metalanguage describes a language by symbols and rules. Symbols can be the following:

- *Terminal symbols* can be keywords (written in **bold** type), key symbols (written in `true type` font e.g., `:=` and `[]`), or generic terminal symbols representing identifiers (written as single letters in *italic* font). Table 4.1 lists the generic terminal symbols used throughout this chapter.

- *Non-terminal symbols* (written as single letters in *italic* font) represent those constructs defined by the rules. Table 4.2 lists the non-terminal symbols used throughout this chapter.

In our variant of EBNF, each non-terminal symbol H is defined by exacly one rule of the form $H ::= L$, where L is a combination of terminal and non-terminal symbols, using the following notation:

- Sequences are represented by a mere succession of symbols.

C: *constructor identifier*	M: *module identifier*	u : *unit identifier*
F: *function identifier*	s : *state identifier*	V: *variable identifier*
G: *gate identifier*	T : *type identifier*	

Table 4.1: Meaning of generic terminal symbols

A: *action*	N: *cardinality list*	U : *unit*
B: *visibility specifier*	O: *communication offer*	W: *time window*
D: *type definition*	P: *pattern*	X : *module (axiom)*
E: *expression*	Q: *semantic modality*	Y : *function definition*
K: *synchronization formula*	R: *synchronizer*	

Table 4.2: Meaning of non-terminal symbols

- Alternatives are separated by vertical bars in normal font ("|").
- Optional parts are enclosed between square brackets in normal font ("[", "]").
- Let $\Box$ be a key symbol (e.g., a comma) or a blank space and $n \in \mathbb{N}$. Then $\alpha_1\Box\ldots\Box\alpha_n$ represents the repetition n times of the symbol α, separated by $\Box$s, and $\alpha_0\Box\ldots\Box\alpha_n$ represents the repetition $n+1$ (i.e., at least one) times of the symbol α, separated by $\Box$s. Instead of n, another letter may designate the maximal index.

Sometimes, we recall a rule given before to discuss only a segment of L. In this case, we use centered dots "$\cdots$" to indicate omissions.

For convenience, we will not always make a strict separation between *objects* and *identifiers* of objects i.e., instead of writing "a module X_1 with the identifier M_1", we will simply write "a module M_1". This lack of precision is common usage and of little practical impact (one exception is discussed in Remark 4.15 on page 103).

Semantics definitions. The semantic rules in this chapter will be presented in Plotkin's SOS (*Structural Operational Semantics*) style [105]. It consists in rules divided by a horizontal line into an upper and a lower part, where the upper part describes zero, one, or several hypotheses, and the lower part one transition derivable from these hypotheses.

4.2 Overview of ATLANTIF

The complete syntax of Atlantif is summarized in Table 4.3. The remainder of this section will contain informal explanations of the constructs. Formal definitions will then be given along the remainder of this chapter.

Syntax of modules:	
X ::= **module** M **is**	
[(**no** | **discrete** | **dense**) **time**]	(*timing option*)
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	
Syntax of units:	
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	
Syntax of actions:	
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*)
| **stop**	(*unit stop*)
| A_1;A_2	(*sequential composition*)
| **if** E **then** A_1 **else** A_2 **end** [**if**]	(*conditional*)
| **case** E **is** P_0->A_0 | ... | P_n->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*)

Syntax of offers:		*Syntax of expressions:*	
O ::= !E	(*value emission*)	E ::= V	(*variable*)
| ?P	(*value reception*)	| $F(E_1, \ldots, E_n)$	(*function*)
		| $C(E_1, \ldots, E_n)$	(*constructor*)

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

Syntax of time windows:	
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 interval*)
Syntax of synchronizers:	
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	

Auxiliary syntax of synchronizers:			
K ::= u	(*single unit*)	N ::= n	(*natural integer*)
| K_1 **and** K_2	(*synchronization*)	| N_1 **or** N_2	(*choice*)
| K_1 **or** K_2	(*alternative*)		
| N **among** ($K_1, \ldots, K_m$)		B ::= **visible**	| **hidden**
| (K_0)		| **urgent**	| **silent**

Table 4.3: Complete ATLANTIF syntax

An ATLANTIF specification is called a *module*. In its header, a module defines whether it is untimed or timed and in the latter case, which kind of time domain is used, namely dense (corresponding to $\mathbb{R}_{\geq 0}$) or discrete (corresponding to $\mathbb{N}$). The body of a module lists, among other things, the two central kinds of constructs, called *units* and *synchronizers*.

Each unit represents one sequential process. It can contain *subunits*, representing subprocesses, thus enabling a description of the unit with a higher granularity. A parallel composition can be modelled by several units being *active* at the same time, where a unit is called active if it is being executed, or *inactive* otherwise. The *initial units* are those units active as soon as the execution of the module begins.

The syntax of a unit is an extension of the syntax of an NTIF process. A unit consists of local variables (that can be shared with its subunits) and a list of *discrete states*, the first of which is the initial state for this unit. To each discrete state is associated an action, built using high-level language constructs and describing data manipulation (deterministic and nondeterministic assignments, variable resets), gate communications, time delays, and jumps to other discrete states, combined by sequential composition, deterministic and nondeterministic choice, and loops. The action associated to a discrete state thus defines a so-called *multibranch transition* that describes several different paths, each of them representing one possible evolution of the unit from the current discrete state to another discrete state.

Time in units is controlled by several new constructs:

- A delay action that was not part of NTIF enables the elapsing of a certain amount of time.
- Life reducers are associated to gate communications, in the form of so-called *time windows*, expressing by one or several intervals how much time can elapse between arriving at this action and executing it. This direct representation of timing constraints makes it unnecessary to introduce clocks.
- The modalities **may** and **must** are associated to gate communications to express whether the time window has a weak or a strong deadline i.e., whether the communication is optional or necessary.

In the syntax of actions, complex compositions are bracketed by keywords, such as **if** $\cdots$ **end if** or **select** $\cdots$ **end select**. The repetition of the initial keyword following **end**, helpful for the readability of nested actions, is optional in these constructs.

Each synchronizer represents the communication constraint associated to a given gate, where the communication is, as the name indicates, a synchronization communication (instead of e.g., a buffer communication). A synchronizer contains a single formula that expresses which sets of units may synchronize. The approach to define this by a separate and expressive construct is inspired by synchronization vectors. Additionally, a synchronizer defines whether a synchronization will be visible, hidden (a τ-transition), or silent i.e., not appear in the semantics at all. If silent, it can also be urgent i.e., it can prohibit any idling when a synchornization on its gate is possible. Finally, synchronizers can stop active units and start inactive units.

In gate communications, a list of *offers* enables data exchange between units. Each offer is either an emission defined by an expression or a reception defined by a pattern.

Identifiers. ATLANTIF identifiers are regular expressions built from uppercase and lowercase letters, digits, and underscores. They must begin with a letter and are not allowed to end with an underscore. E.g., "*State1*" and "*ONE_two_3*" are valid identifiers, but "*4_a*", "*Seti@Home*", and "*A_2_*" are not.

Notation. For a given module, we write $\mathcal{V}$ the set of variable identifiers, $\mathbb{G}$ the set of gates, and $\mathbb{U}$ the set of unit identifiers. The subsets of $\mathbb{U}$ are written $\mathcal{U}, \mathcal{U}', \mathcal{U}_0, \mathcal{U}_1$, etc.

We also write $\mathcal{S}$ the set of discrete state identifiers. $\mathcal{S}$ contains two special elements written δ and Ω, reserved for the semantics definitions. The element δ denotes the termination of an action, thus enabling the execution of subsequent actions. The element Ω denotes the termination of a unit. At last, for a given unit u, we write $\mathcal{S}_u$ the set containing all discrete state identifiers used in u, as well as δ and Ω.

Example 4.1. *We still consider the behaviour described in Example 3.1 by two timed automata and give in Fig. 4.1 the equivalent in* ATLANTIF.

```
module Light is
dense time
sync Push is User and Lamp end sync
init Lamp, User

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

unit User is
  from Dozing
    Push;
    select to Phoning
        [] to Prepare_Work
    end select

  from Phoning
    wait 1000; Push; to Dozing

  from Prepare_Work
    wait 1; Push must in [0,2];
    to Working

  from Working
    wait 1000; Push; to Dozing
end unit

end module
```

Figure 4.1: ATLANTIF program describing a light switch

4.3 Basic constructs

4.3.1 Types, functions, and constructors

As an intermediate model, Atlantif must be able to express standard data types, such as booleans, integers, floating-point numbers, etc., and constructed data types, such as lists, records, arrays, etc. Also, there should be the possibility of defining functions to manipulate data.

We consider types and functions to be out of the scope of this work, because they have already been the subject of related work (cf. e.g., Lotos NT [115] or Fiacre [24]). Also, to a large extend, they are orthogonal to concurrent and real-time aspects, thus it is possible to restrict them here to a minimum without harming the consistency of the language[10]. Thus, the syntax and semantics of types, functions, and constructors presented here only covers the minimum needed for the following sections.

Syntax description. *Type declarations* are of the form "**type** T **is** D", where T is an identifier and D a type definition (not detailed). We suppose the existence of the predefined types "**bool**", "**int**", and "**float**" in their usual sense. Each type T is defined by one or several *constructors*, which are used to create instances T. Each constructor C is defined by a (possibly empty) list of types, corresponding to the parameters that are used by C.

Function declarations are of the form "**function** F **is** Y", where F is an identifier and Y a function definition. The function definition is not detailed here, but we suppose it describes a function with a prototype of the form "$T_1 \times \ldots \times T_n \rightarrow T$", where $T_1, \ldots, T_n$ are the parameter types and T the return type. We suppose the existence of predefined arithmetical functions such as "+", "−", and "∗" and boolean operators such as "=", "≥", and "<", defined on the predefined types with their usual meaning. For convenience, we will use the usual infix notation for these predefined functions.

4.3.2 Expressions

$$
\begin{array}{lll}
E ::= & V & (\textit{variable}) \\
 & \mid F\,(E_1, \ldots, E_n) & (\textit{function expression}) \\
 & \mid C\,(E_1, \ldots, E_n) & (\textit{constructor expression})
\end{array}
$$

Note that this definition includes constant values as the special case of constructor calls without parameters.

Static semantics. An expression E is *well-typed* if it can be given a unique type. We define a function *type* on the set of producible expressions and we write $\textit{type}(E) = T$ if

[10] In Section 3.5.1, we discussed exceptions to this orthogonality. Therefore, the indicated "minimum" covers our solutions regarding these exceptions.

one of the following is true:

- E consists of a variable of type T.
- E is a function expression of the form $F\texttt{(}E_1\texttt{,} \ldots \texttt{,}E_n\texttt{)}$, where F has the prototype $T_1 \times \ldots \times T_n \to T$ and for each $i \in 1..n$, $\mathit{type}(E_i) = T_i$.
- E is a constructor expression of the form $C\texttt{(}E_1\texttt{,} \ldots \texttt{,}E_n\texttt{)}$, where C is a constructor of T with parameters of respective types "$T_1, \ldots, T_n$" and for each $i \in 1..n$, $\mathit{type}(E_i) = T_i$.

For each expression E, we define the set $\mathit{use}(E)$ containing all variables that need to have a value assigned before an evaluation of E as follows:

$$\begin{aligned}
\mathit{use}(V) &\stackrel{\text{def}}{=} \{V\} \\
\mathit{use}(F\texttt{(}E_1\texttt{,} \ldots \texttt{,}E_n\texttt{)}) &\stackrel{\text{def}}{=} \textstyle\bigcup_{i\in 1..n} \mathit{use}(E_i) \\
\mathit{use}(C\texttt{(}E_1\texttt{,} \ldots \texttt{,}E_n\texttt{)}) &\stackrel{\text{def}}{=} \textstyle\bigcup_{i\in 1..n} \mathit{use}(E_i)
\end{aligned}$$

We also define the set $\mathit{read}(E)$ containing all variables that are read in an evaluation of E as follows:

$$\mathit{read}(E) \stackrel{\text{def}}{=} \mathit{use}(E)$$

Dynamic semantics. We assume a set *Val* containing all *values* defined by the declared types. Values are written v, v', v_0, v_1, etc. The values of variables are given by partial functions on $\mathcal{V} \to \mathit{Val}$, called *stores* and written $\rho, \rho', \rho_0, \rho_1$, etc.

The semantics of expressions is given by a predicate $\mathit{eval}(E, \rho, v)$ that is **true** if and only if the evaluation of expression E in store ρ yields a value v. Formally:

$$\mathit{eval}(E, \rho, v) \text{ iff } \begin{cases}
E = V \text{ and } \rho(V) = v \\
E = C\texttt{(}E_1\texttt{,} \ldots \texttt{,}E_n\texttt{)},\ \mathit{eval}(E_1, \rho, v_1), \ldots, \mathit{eval}(E_n, \rho, v_n), \\
\qquad\quad \text{and } C(v_1, \ldots, v_n) = v \\
E = F\texttt{(}E_1\texttt{,} \ldots \texttt{,}E_n\texttt{)},\ \mathit{eval}(E_1, \rho, v_1), \ldots, \mathit{eval}(E_n, \rho, v_n), \\
\qquad\quad \text{and } F(v_1, \ldots, v_n) = v
\end{cases}$$

Remark 4.1. *Functions are well-defined i.e., deterministic. For each expression E, store ρ and values v_1, v_2, we have: If $\mathit{eval}(E, \rho, v_1)$ and $\mathit{eval}(E, \rho, v_2)$, then $v_1 = v_2$.*

4.3.3 Patterns

Values can be assigned to patterns using pattern-matching. A value can be matched against a pattern if both the value and the pattern have the same shape. In that case, the variables contained in the pattern get assigned with appropriate values. Note that in the case of a constructor pattern with several parameters, pattern-matching operates sequentially, from the left to the right.

$$
\begin{array}{rll}
P ::= & V & (\textit{variable}) \\
\mid & \textbf{any } T & (\textit{anonymous variable}) \\
\mid & C\,\texttt{(}P_1\texttt{,} \ldots \texttt{,}P_n\texttt{)} & (\textit{constructor pattern}) \\
\mid & P_0 \textbf{ where } E & (\textit{conditional pattern}) \\
\mid & \texttt{(}P_0\texttt{)} &
\end{array}
$$

Static semantics. A pattern P is well-typed if it can be given a unique type. We write $\mathit{type}(P) = T$ if one of the following is true:

- P consists of a variable V, where $\mathit{type}(V) = T$.
- P consists of an anonymous variable **any** T and T is a type.
- P is a constructor pattern of the form $C\,\texttt{(}P_1\texttt{,} \ldots \texttt{,}P_n\texttt{)}$, where C is a constructor of a type T, with parameters of respective types $T_1, \ldots, T_n$, and for each $i \in 1..n$, $\mathit{type}(P_i) = T_i$.
- P is a conditional pattern of the form P_0 **where** E, where $\mathit{type}(P_0) = T$ and $\mathit{type}(E) = \textbf{bool}$.

For each pattern P, we define the set $\mathit{use}(P)$ containing all variables that need to have a value assigned before a pattern-matching against P, along with a set $\mathit{def}(P)$ containing all variables that are necessarily assigned a value after a pattern-matching against P as follows:

$$
\begin{array}{ll}
\mathit{use}(V) \stackrel{\text{def}}{=} \varnothing & \mathit{def}(V) \stackrel{\text{def}}{=} \{V\} \\
\mathit{use}(\textbf{any } T) \stackrel{\text{def}}{=} \varnothing & \mathit{def}(\textbf{any } T) \stackrel{\text{def}}{=} \varnothing \\
\mathit{use}(C(P_1, \ldots, P_n)) \stackrel{\text{def}}{=} \bigcup_{i \in 1..n} \mathit{use}(P_i) & \mathit{def}(C(P_1, \ldots, P_n)) \stackrel{\text{def}}{=} \bigcup_{i \in 1..n} \mathit{def}(P_i) \\
\mathit{use}(P_0 \textbf{ where } E) \stackrel{\text{def}}{=} & \mathit{def}(P_0 \textbf{ where } E) \stackrel{\text{def}}{=} \mathit{def}(P_0) \\
\qquad (\mathit{use}(E) \setminus \mathit{def}(P_0)) \cup \mathit{use}(P_0) &
\end{array}
$$

We also define the set $\mathit{read}(P)$ containing all variables that may be read during a pattern-matching against P, and the set $\mathit{write}(P)$ containing all variables that may be written during a pattern-matching against P as follows:

$$
\begin{array}{ll}
\mathit{read}(V) \stackrel{\text{def}}{=} \varnothing & \mathit{write}(P) \stackrel{\text{def}}{=} \mathit{def}(P) \\
\mathit{read}(\textbf{any } T) \stackrel{\text{def}}{=} \varnothing & \\
\mathit{read}(C(P_1, \ldots, P_n)) \stackrel{\text{def}}{=} \bigcup_{i \in 1..n} \mathit{read}(P_i) & \\
\mathit{read}(P_0 \textbf{ where } E) \stackrel{\text{def}}{=} \mathit{read}(E) \cup \mathit{read}(P_0) &
\end{array}
$$

Each pattern has to be *well-bound* i.e., the variables defined by this pattern have to be assigned unambiguously. Patterns of the form V and **any** T are always well-bound. "P_0 **where** E" is well-bound if P_0 is well-bound. $C(P_1, \ldots, P_n)$ is well-bound if every variable is defined at most once in $P_1, \ldots, P_n$ and if no variable is used before being defined, patterns being matched from left to right. Formally, a constructor pattern of the form $C(P_1, \ldots, P_n)$ is well-bound if the following are all satisfied:

$$
\begin{array}{ll}
(i) & (\forall i \in 1..n)\ P_i \text{ is well-bound} \\
(ii) & (\forall i, j \in 1..n, i \neq j)\ \mathit{def}(P_i) \cap \mathit{def}(P_j) = \varnothing \\
(iii) & (\forall i, j \in 1..n, i < j)\ \mathit{use}(P_i) \cap \mathit{def}(P_j) = \varnothing
\end{array}
$$

For instance, given a binary constructor C, and two variables V_1, V_2, this means:

- $C(V_1, V_1)$ is not well-bound, because it twice assigns a value to V_1 i.e., the value of the second assignment destroys the value of the first assignment. Neither is $C(V_1 \textbf{ where } V_1 = V_2, V_2)$ well-bound, because the condition $V_1 = V_2$ is evaluated before the assignment of V_2. Thus, the condition is ambiguous.
- $C(V_1, V_2 \textbf{ where } V_1 = V_2)$ and "$C(V_1, V_2) \textbf{ where } V_1 = V_2$" are well-bound, as their conditions do not depend on variables that are undefined or that may change value during further pattern-matching: In both cases, both V_1 and V_2 are assigned a new value *before* being compared.

Dynamic semantics. We define a *pattern-matching* function $\mathit{match}(v, \rho, P)$ that returns either **fail** if the value v does not match the pattern P in a context defined by the store ρ, 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. Formally:

$$
\begin{array}{l}
\mathit{match}(v, \rho, V) \stackrel{\text{def}}{=} \rho \oslash [V \mapsto v] \\
\mathit{match}(v, \rho, \textbf{any } T) \stackrel{\text{def}}{=} \rho \\
\mathit{match}(C(v_1, \ldots, v_n), \rho_0, C(P_1, \ldots, P_n)) \stackrel{\text{def}}{=} \begin{cases} \rho_n & \text{if } (\exists \rho_0, \ldots, \rho_n)\ (\forall i \in 1..n) \\ & \qquad \mathit{match}(v_i, \rho_{i-1}, P_i) = \rho_i \neq \textbf{fail} \\ \textbf{fail} & \text{otherwise} \end{cases} \\
\mathit{match}(v, \rho, P_0 \textbf{ where } E) \stackrel{\text{def}}{=} \begin{cases} \rho' & \text{if } \mathit{match}(v, \rho, P_0) = \rho' \neq \textbf{fail} \text{ and } \mathit{eval}(E, \rho', \textbf{true}) \\ \textbf{fail} & \text{otherwise} \end{cases} \\
\mathit{match}(v, \rho, P) \stackrel{\text{def}}{=} \textbf{fail} \qquad \text{if none of the above conditions hold}
\end{array}
$$

4.3.4 Offers

Value exchanges between processes are expressed by *offers*, which have the following form:

$$\begin{array}{rll} O ::= & \texttt{!}E & (\textit{emission of the value of } E) \\ \mid & \texttt{?}P & (\textit{reception of a value matching } P) \end{array}$$

Static semantics. An offer O is well-typed if it can be given a unique type. We write $\mathit{type}(O) = T$ if one of the following is true:

- O is an emission offer of the form $\texttt{!}E$ and $\mathit{type}(E) = T$.
- O is a reception offer of the form $\texttt{?}P$ and $\mathit{type}(P) = T$.

The sets *use*, *def*, *read*, and *write* defined above for patterns and expressions are extended to offers as follows:

$$\begin{array}{llll} \mathit{use}(\texttt{!}E) \stackrel{\text{def}}{=} \mathit{use}(E) & \mathit{def}(\texttt{!}E) \stackrel{\text{def}}{=} \varnothing & \mathit{read}(\texttt{!}E) \stackrel{\text{def}}{=} \mathit{read}(E) & \mathit{write}(\texttt{!}E) \stackrel{\text{def}}{=} \varnothing \\ \mathit{use}(\texttt{?}P) \stackrel{\text{def}}{=} \mathit{use}(P) & \mathit{def}(\texttt{?}P) \stackrel{\text{def}}{=} \mathit{def}(P) & \mathit{read}(\texttt{?}P) \stackrel{\text{def}}{=} \mathit{read}(P) & \mathit{write}(\texttt{?}P) \stackrel{\text{def}}{=} \mathit{write}(P) \end{array}$$

Dynamic semantics. The semantics of an offer O is given by a function $\mathit{accept}(v, \rho, O)$ that returns either **fail** if in a context given by the store ρ, the value v does not match the offer O, or a new store ρ' otherwise. Formally:

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

4.4 Units

4.4.1 Overview

Each process of a system is represented by a *unit*, using the following syntax:

$$\begin{array}{lll} U ::= & \textbf{unit } u \textbf{ is} & \\ & [\textbf{variables } V_0 \texttt{:} T_0 \ [\texttt{:=}E_0]\texttt{,} \ldots \texttt{,} V_n \texttt{:} T_n \ [\texttt{:=}E_n]] & (\textit{variable declarations}) \\ & \textbf{from } s_0 \ A_0 \ \ldots \ \textbf{from } s_m \ A_m & (\textit{list of multibranch transitions}) \\ & U_1 \ \ldots \ U_l & (\textit{subunits}) \\ & \textbf{end unit} & \end{array}$$

The variables used in a unit have to be declared a type, and an optional initial value. We write $\mathit{decl}(u)$ the set of variables declared in u.

Static semantics. Types with the identifiers $T_0, \ldots, T_n$ have to be declared. The identifiers $V_0, \ldots, V_n$ have to be pairwise distinct. The declarations have to be *well-typed* (cf. Section 4.3.2) i.e., the optional initial values must be well-typed constant expressions of the corresponding type.

The list of *multibranch transitions* has the form "**from** s_0 A_0 ... **from** s_m A_m", where $s_0, \ldots, s_m$ are the *discrete states* of the unit and with each discrete state s_i ($i \in 0..m$) is associated an action A_i. We define the function *act* by $act(s_i) \stackrel{\text{def}}{=} A_i$ and $act(\Omega) \stackrel{\text{def}}{=}$ **null** ($act(\delta)$ is not defined). The discrete state s_0 is the initial state of u.

4.4.2 Actions

ATLANTIF inherits the *action* syntax of the intermediate format NTIF [61], and extends it with real-time constructs. The syntax of actions is given as follows, where shading indicates additions that were made in ATLANTIF with respect to NTIF:

$A ::=$	$V_0, \ldots, V_n$:= $E_0, \ldots, E_n$	(*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*)
	| **stop**	(*unit stop*)
	| $A_1 ; A_2$	(*sequential composition*)
	| **if** E **then** A_1 **else** A_2 **end** [**if**]	(*conditional branching*)
	| **case** E **is** P_0->A_0 | ... | P_n->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*)

In the remainder of this section, we will discuss each action separately.

Static semantics. An action has to satisfy *well-binding* i.e., variable assignments made along this action have to be unambiguous. It also has to satisfy well-typing. Details on these two criteria are given with each action description in the remainder of this section.

A variable V may be read during an evaluation of an action A if A contains an expression containing V; V may be written during an evaluation of A if A contains a pattern writing V or an assignment to V. In each action description below, the formal definition of the corresponding sets *read*(A) and *write*(A) will be given.

Dynamic semantics. For the semantics of actions, we define the set of *local labels* $\mathbb{L}_1 \stackrel{\text{def}}{=} \{G\ v_1 \ldots v_n \mid G \in \mathbb{G},\ v_1, \ldots, v_n \in Val\} \cup \{\epsilon\}$. ε is a special label, which represents transitions without communication actions.

The dynamic semantics of actions describes transitions between one action towards one discrete state. In NTIF, such transitions were 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 [61]. ATLANTIF extends this to a relation of the form $(A, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho')$, where (s, d', ρ') is called a *local state* (also written $\sigma, \sigma', \sigma_0, \sigma_1$, etc.) and d, d' are tuples of the form (t, μ), described as follows:

- $t \in \mathbb{D}$ is called a *phase*. Intuitively, the phase represents the time difference between the local state of a unit and the global semantic state of the module (which we will define in Section 4.6.3). From the perspective of within a unit, the phase therefore corresponds to the time that may elapse in this unit until the next communication.

  Technically, the phase indicates how much time has elapsed (in the whole module) since the unit made its last (non-**silent**) communication. This value is reduced by the values of all **wait** actions, that were executed since then.

- $\mu \in \{\textbf{true}, \textbf{false}\}$ is called a *blocking condition*. It is equal to **true** if and only if time is not allowed to elapse further than given by the phase t, thus representing a strong deadline.

"$(A, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho')$" means that the action A in the context defined by d and ρ evolves, in a transition labelled by l, to the local state (s, d', ρ').

The formal definitions are given below with each action.

Deterministic assignment

$$\begin{array}{rl} A ::= & V_0\texttt{,} \ldots \texttt{,}V_n \texttt{ := } E_0\texttt{,} \ldots \texttt{,}E_n \\ | & \ldots \end{array}$$

This action simultaneously assigns the values of $E_0, \ldots, E_n$ to $V_0, \ldots, V_n$ respectively.

Static semantics. The deterministic assignment action is well-bound if the variables $V_0, \ldots, V_n$ are pairwise distinct. It is well-typed if for each $i \in 0..n$, $\mathit{type}(V_i) = \mathit{type}(E_i)$. The sets *read* and *write* are defined as follows:

$$\begin{array}{l} \mathit{read}(V_0\texttt{,} \ldots \texttt{,}V_n \texttt{ := } E_0\texttt{,} \ldots \texttt{,}E_n) \stackrel{\text{def}}{=} \bigcup_{i \in 0..n} \mathit{read}(E_i) \\ \mathit{write}(V_0\texttt{,} \ldots \texttt{,}V_n \texttt{ := } E_0\texttt{,} \ldots \texttt{,}E_n) \stackrel{\text{def}}{=} \{V_0, \ldots, V_n\} \end{array}$$

Dynamic semantics. Deterministic assignments define local transitions as follows:

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

Note that the semantics indeed describes a simultaneous assignment. For instance, in "V_1, V_2 := V_2, V_1", the two variables swap their values, which is obviously different from a sequence of two assignments, namely "V_1 := V_2" followed by "V_2 := V_1".

Nondeterministic assignment

When a system cannot be specified in all details, abstractions have to be made, for instance by the introduction of nondeterminism in variable assignments.

$$\begin{array}{rl} A ::= & \cdots \\ | & V_0\,,\ldots,V_n \;\texttt{:=}\; \textbf{any}\; T_0\,,\ldots,T_n\; [\textbf{where}\; E] \\ | & \cdots \end{array}$$

This action assigns to each variable $V_0, \ldots, V_n$ an arbitrary value so that the condition E evaluates to **true**. If unspecified, we suppose E to be of the form **true**.

Static semantics. The nondeterministic assignment action is well-bound if the variables $V_0, \ldots, V_n$ are pairwise distinct. It is well-typed if $\mathit{type}(E) = \textbf{bool}$ and for each $i \in 0..n$, $\mathit{type}(V_i) = T_i$. Remark 4.14 on page 103 discusses further restrictions on E. The sets *read* and *write* are defined as follows:

$$\begin{aligned} &\mathit{read}(V_0\,,\ldots,V_n \;\texttt{:=}\; \textbf{any}\; T_0\,,\ldots,T_n\; \textbf{where}\; E) \stackrel{\text{def}}{=} \mathit{read}(E) \\ &\mathit{write}(V_0\,,\ldots,V_n \;\texttt{:=}\; \textbf{any}\; T_0\,,\ldots,T_n\; \textbf{where}\; E) \stackrel{\text{def}}{=} \{V_0, \ldots, V_n\} \end{aligned}$$

Dynamic semantics. To each variable V_i ($i \in 0..n$) a new value of type T_i is assigned, such that the condition E (possibly depending on $V_0, \ldots, V_n$) is satisfied.

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

Variable reset

$$\begin{array}{rl} A ::= & \cdots \\ | & \textbf{reset}\; V_0\,,\ldots,V_n \\ | & \cdots \end{array}$$

A variable reset performs the un-assignment of the values currently assigned to the variables $V_0, \ldots, V_n$. This can for instance be used to emulate variables with a limited scope: The **reset** action then represents the end of this scope, after which the variables become unavailable.

Static semantics. A reset action is always well-bound and well-typed. The sets *read* and *write* are defined as follows:

$$\mathit{read}(\textbf{reset } V_0\textbf{,} \ldots \textbf{,} V_n) \stackrel{\text{def}}{=} \varnothing \qquad \mathit{write}(\textbf{reset } V_0\textbf{,} \ldots \textbf{,} V_n) \stackrel{\text{def}}{=} \varnothing$$

Dynamic semantics. Reset actions define local transitions as follows:

$$(\mathit{reset})\ \frac{}{(\textbf{reset } V_0, \ldots, V_n, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, d, \rho \ominus \{V_0, \ldots, V_n\})}$$

Note that a variable among $V_0, \ldots, V_n$ may already have no value assigned, in which case nothing happens to this variable. Note also that the ATLANTIF **reset** is an operation different from the "clock reset" of timed automata (cf. Section 3.2.2), which does not remove values, but replaces them by zero.

Delay

$$\begin{array}{rcl} A ::= & & \cdots \\ & | & \textbf{wait } E \\ & | & \cdots \end{array}$$

The **wait** action implements a time delay (of E time units) that is independent of communication actions. It thus provides an intuitive means to represent e.g., processing time within one unit. Such a construct appears in similar shapes in different formalisms, such as in TCSP [108].

Static semantics. Delay actions must not occur in modules with the timing option **no time**.

A delay action is always well-bound. It is well-typed if $\mathit{type}(E) = \textbf{int}$ in **discrete time** or $\mathit{type}(E) = \textbf{float}$ in **dense time**. The sets *read* and *write* are defined as follows:

$$\mathit{read}(\textbf{wait } E) \stackrel{\text{def}}{=} \mathit{read}(E) \qquad \mathit{write}(\textbf{wait } E) \stackrel{\text{def}}{=} \varnothing$$

Dynamic semantics. Delays define local transitions as follows:

$$(\mathit{wait})\ \frac{\mathit{eval}(E, \rho, v) \wedge t \geq v \geq 0}{(\textbf{wait } E, (t, \mu), \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, (t - v, \mu), \rho)}$$

This rule is the only one in the action semantics that modifies the phase t of a local state. Recalling that the phase t indicates how much the unit differs from other units regarding time elapsing, then it is clear that this phase is updated by a delay in this single unit.

Gate communication

$$
\begin{array}{rll}
A ::= & \cdots & \\
| & G\ O_1 \ldots O_n\ [[Q]\ \textbf{in}\ W] & \\
| & \cdots & \\
W ::= & \texttt{[}E_1\texttt{,}E_2\texttt{]} \quad | \quad \texttt{]}E_1\texttt{,}E_2\texttt{]} \quad | \quad \texttt{[}E_1\texttt{,}E_2\texttt{[} \quad | \quad \texttt{]}E_1\texttt{,}E_2\texttt{[} & (\textit{bounded interval}) \\
| & \texttt{[}E_1\texttt{, ...[} \quad | \quad \texttt{]}E_1\texttt{, ...[} & (\textit{unbounded interval}) \\
| & W_1\ \textbf{or}\ W_2 \ | \ W_1\ \textbf{and}\ W_2 \ | \ \texttt{(}W_0\texttt{)} & (\textit{combined intervals}) \\
Q ::= & \textbf{must} \ | \ \textbf{may} &
\end{array}
$$

This action invokes a gate communication, whose syntax components are as follows:

- G is the gate on which the synchronization is made.
- $O_1, \ldots, O_n$ are the offers (cf. Section 4.3.4). A combination of emission offers ($\texttt{!}E$) and reception offers ($\texttt{?}P$) is possible.
- W is a *time window*, defined using intersections (**and**) and unions (**or**) of open or closed intervals, where "`...`" represents infinity and **and** binds more strongly than **or**[11]. 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,...[`" (i.e., no restriction is assumed).
- Q is a *modality*. **must** indicates that the communication must occur before the end of the time window (which is called the *deadline*), and **may** indicates that time can elapse indefinitely, without the communication ever occurring. If unspecified, Q is taken to be **may**. In our classification inspired by [36], **may** corresponds to a *weak* deadline, whereas **must** corresponds to a *strong* deadline.

Static semantics. If the modality is **must**, then all intervals contained in W must either be unbounded (i.e., have a right-hand side of the form "`,...[`") or right-closed (i.e., have a right-hand side of the form "$\texttt{,}E_2\texttt{]}$"). This restriction ensures the correct functioning of the blocking condition (cf. page 62), which becomes **true** when the phase of the unit has reached the deadline of a communication with **must** modality. Without this restriction, such a limit could not be defined. In particular, this restriction is coherent with the intuitive idea that at any moment, the maximal amount of time that can elapse can be determined. Further static semantics restrictions on time windows depend on the type of the synchronizer and will be defined in Section 4.6.2.

The set *read* is extended to time windows as follows:

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

[11] I.e., "W_1 **and** W_2 **or** W_3" equals "$\texttt{(}W_1$ **and** $W_2\texttt{)}$ **or** W_3".

Well-binding for a communication action $G\ O_1 \ldots O_n\ Q$ **in** W requires that variable assignments in offers are unambiguous and that the time window must not depend on the offers. Formally:

$$
\begin{array}{ll}
(i) & (\forall i \in 1..n) \text{ If } O_i \text{ has the form } ?P\text{, then } P \text{ has to be well-bound} \\
(ii) & (\forall i, j \in 1..n, i \neq j)\ \mathit{def}(O_i) \cap \mathit{def}(O_j) = \varnothing \\
(iii) & (\forall i, j \in 1..n, i < j)\ \mathit{use}(O_i) \cap \mathit{def}(O_j) = \varnothing \\
(iv) & \left(\bigcup_{i \in 1..n} \mathit{def}(O_i)\right) \cap \mathit{read}(W) = \varnothing
\end{array}
$$

Restriction *(iv)* prohibits "pathological" code like "$G\ ?V$ **must in** `[0,`V`]`". Our semantics definition depends on determining for each time window with the modality **must** how much time to elapse it permits, which is not possible when the time window depends on offers. Moreover, we consider time windows depending on offers as unintuitive, because the question of whether the communication action is available or not would depend on a data reception in the future and could thus not be answered before the communication is executed. Also, it is not clear in what cases such code would be useful.

A communication action is well-typed if each offer is well-typed and if each expression appearing in W has type **int** in **discrete time** or type **float** in **dense time**. If the timing option is "**no time**", time windows must not be used. The sets *read* and *write* are defined on communications as follows:

$$
\begin{array}{l}
\mathit{read}(G\ O_1 \ldots O_n\ Q \textbf{ in } W) \stackrel{\text{def}}{=} \mathit{read}(W) \cup \bigcup_{i \in 1..n} \mathit{read}(O_i) \\
\mathit{write}(G\ O_1 \ldots O_n\ Q \textbf{ in } W) \stackrel{\text{def}}{=} \bigcup_{i \in 1..n} \mathit{write}(O_i)
\end{array}
$$

Dynamic semantics. The semantics of a time window is defined by the predicate $\mathit{win\_eval}(W, \rho, D)$ that is **true** if and only if the evaluation of W in store ρ yields a (possibly infinite) set of time instants D, where D is a subset of the time domain $\mathbb{D}$. Formally:

$$
\begin{array}{l}
\mathit{win\_eval}(\texttt{[}E_1\texttt{,}E_2\texttt{]}, \rho, D) \stackrel{\text{def}}{=} \\
\qquad\qquad (\exists v_1, v_2)\ \mathit{eval}(E_1, \rho, v_1) \text{ and } \mathit{eval}(E_2, \rho, v_2) \text{ and } D = [v_1, v_2] \cap \mathbb{D} \\
\qquad \text{(similarly for } \texttt{[}E_1\texttt{,}E_2\texttt{[}, \texttt{]}E_1\texttt{,}E_2\texttt{]}, \text{ and } \texttt{]}E_1\texttt{,}E_2\texttt{[}) \\
\mathit{win\_eval}(\texttt{[}E_1\texttt{, ...[}, \rho, D) \stackrel{\text{def}}{=} \\
\qquad\qquad (\exists v_1)\ \mathit{eval}(E_1, \rho, v_1) \text{ and } D = \mathbb{D} \setminus ([0, v_1[) \\
\qquad \text{(similarly for } \texttt{]}E_1\texttt{, ...[}) \\
\mathit{win\_eval}(W_1 \textbf{ and } W_2, \rho, D) \stackrel{\text{def}}{=} \\
\qquad\qquad (\exists D_1, D_2)\ \mathit{win\_eval}(W_1, \rho, D_1) \text{ and } \mathit{win\_eval}(W_2, \rho, D_2) \text{ and } D = D_1 \cap D_2 \\
\mathit{win\_eval}(W_1 \textbf{ or } W_2, \rho, D) \stackrel{\text{def}}{=} \\
\qquad\qquad (\exists D_1, D_2)\ \mathit{win\_eval}(W_1, \rho, D_1) \text{ and } \mathit{win\_eval}(W_2, \rho, D_2) \text{ and } D = D_1 \cup D_2
\end{array}
$$

The time window thus has the role of a *life reducer*, similar to that found in different timed process algebras such as RT-Lotos [47].

We also define a predicate $up\_lim(Q, D, t)$ that is **true** if and only if $Q =$ **must** and the set D has a maximum equal to t[12]. This means that t time units in the future will be the last moment when the communication can be performed.

Because of Remark 4.1 on page 57 (i.e., in an expression $eval(E, \rho, v)$, v is uniquely determined by E and ρ), there is a unique D for given W, ρ in $win\_eval(W, \rho, D)$ and up_lim is thus well-defined.

The semantics of the communication action is given by the following rule:

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

Note that the evaluation of the time window is done with the store ρ_{n+1} instead of ρ_1. Because of the static semantics restriction on windows being independent of offers, this makes no difference. Note also that in a first draft of Atlantif, the timed condition had been realized by a time capture variable and a constraint formula depending on this variable (cf. Section 3.3.3). Given that time capture is less adapted to formal verification [52], we replaced it by a time window.

Jump

$$\begin{array}{rl} A ::= & \cdots \\ | & \textbf{to}\ s \\ | & \cdots \end{array}$$

This action executes a jump to the discrete state s.

Static semantics. A jump action is always well-bound and well-typed. The discrete state s has to be defined among the discrete states of the current unit i.e., $s \in (\mathcal{S}_u \backslash \{\delta, \Omega\})$. The sets *read* and *write* are defined as follows:

$$read(\textbf{to}\ s) \overset{\text{def}}{=} \varnothing \qquad write(\textbf{to}\ s) \overset{\text{def}}{=} \varnothing$$

Dynamic semantics. Jumps define local transitions as follows:

$$(to)\ \frac{}{(\textbf{to}\ s, d, \rho) \overset{\varepsilon}{\Longrightarrow} (s, d, \rho)}$$

[12] This being the only semantic consequence of the **must**, it can be seen that this keyword has no effect if the time window is unbounded.

Stop

$$\begin{array}{rl} A ::= & \cdots \\ | & \mathbf{stop} \\ | & \cdots \end{array}$$

stop ends the sequential behaviour defined by a unit by jumping to the termination state Ω, where the unit cannot perform any communications. Note that this is a regular termination and not a timelock (cf. page 19) state, because the elapsing of time is not affected by a unit being in Ω.

Static semantics. A **stop**-action is always well-bound and well-typed. The sets *read* and *write* are defined as follows:

$$\mathit{read}(\mathbf{stop}) \stackrel{\text{def}}{=} \varnothing \qquad \mathit{write}(\mathbf{stop}) \stackrel{\text{def}}{=} \varnothing$$

Dynamic semantics. Stop actions define local transitions as follows:

$$(\mathit{stop})\ \frac{}{(\mathbf{stop}, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\Omega, d, \rho)}$$

Note that this is the only rule using the special discrete state Ω.

Sequential composition

$$\begin{array}{rl} A ::= & \cdots \\ | & A_1 \,;A_2 \\ | & \cdots \end{array}$$

This action symmetrically composes two actions sequentially: A_2 starts as soon as A_1 terminates, except if A_1 executes a jump to another discrete state or stops, in which case A_2 is ignored. The symmetrical composition is an alternative to the *action-prefix* notation, which restricts the left-hand side of a composition to atomic actions (used in e.g., CSP [77], CCS [96], LOTOS [81]).

The symbol "**;**" is also used for symmetrical sequential composition in other languages, such as LOTOS NT [115] and PROMELA [79].

Static semantics. $A_1 \,; A_2$ is well-bound if both A_1 and A_2 are well-bound, and well-typed if both A_1 and A_2 are well-typed. The sets *read* and *write* are defined as follows:

$$\mathit{read}(A_1 \,; A_2) \stackrel{\text{def}}{=} \mathit{read}(A_1) \cup \mathit{read}(A_1) \qquad \mathit{write}(A_1 \,; A_2) \stackrel{\text{def}}{=} \mathit{write}(A_1) \cup \mathit{write}(A_1)$$

Dynamic semantics. Each of the actions A_1 and A_2 define a transition between local states. Those transitions are merged into a single transition, by using the binary operator $+$ defined as a partial function on $\mathbb{L}_1 \times \mathbb{L}_1 \rightarrow \mathbb{L}_1$ by $l + \varepsilon \stackrel{\text{def}}{=} l$, $\varepsilon + l \stackrel{\text{def}}{=} l$. If both its operands are different from ε, it is undefined[13].

The semantics of sequential composition is then given by the following two rules:

$$(\mathit{seq}_1)\ \frac{(A_1, d, \rho) \stackrel{l_1}{\Longrightarrow} (\delta, d', \rho') \wedge (A_2, d', \rho') \stackrel{l_2}{\Longrightarrow} \sigma}{(A_1\,;A_2, d, \rho) \stackrel{l_1+l_2}{\Longrightarrow} \sigma}$$

$$(\mathit{seq}_2)\ \frac{(A_1, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho') \wedge s \neq \delta}{(A_1\,;A_2, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho')}$$

Nested symmetrical sequential composition can be ambiguous, unless it is associative. We will prove associativity in Proposition 4.5 on page 96.

Deterministic choice

$$\begin{array}{rl} A ::= & \cdots \\ | & \textbf{case } E \textbf{ is } P_0\texttt{->}A_0 \,\texttt{|}\, \ldots \,\texttt{|}\, P_n\texttt{->}A_n \textbf{ end } [\textbf{case}] \\ | & \cdots \end{array}$$

This action performs one of the actions A_i $(i \in 0, \ldots, n)$, if the corresponding pattern P_i (cf. Section 4.3.3) is the first one that matches the expression E.

Static semantics. The **case**-action is well-bound if for each $i \in 0..n$, P_i and A_i are well-bound. It is well-typed if for each $i \in 0..n$, A_i is well-typed and $\mathit{type}(P_i) = \mathit{type}(E)$. The sets *read* and *write* are defined as follows:

$$\mathit{read}(\textbf{case } E \textbf{ is } P_0\texttt{->}A_0 \,\texttt{|}\, \ldots \,\texttt{|}\, P_n\texttt{->}A_n \textbf{ end}) \stackrel{\text{def}}{=} \mathit{read}(E) \cup \textstyle\bigcup_{i \in 0..n}(\mathit{read}(P_i) \cup \mathit{read}(A_i))$$
$$\mathit{write}(\textbf{case } E \textbf{ is } P_0\texttt{->}A_0 \,\texttt{|}\, \ldots \,\texttt{|}\, P_n\texttt{->}A_n \textbf{ end}) \stackrel{\text{def}}{=} \textstyle\bigcup_{i \in 0..n}(\mathit{write}(P_i) \cup \mathit{write}(A_i))$$

Dynamic semantics. The patterns P_0 to P_n are evaluated one after another, until a P_i is found to match the expression E. Then, the action A_i is executed.

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

[13] In Section 4.4.3 however, we will introduce a static semantics restriction that guarantees that one of the operands equals ε.

As the static semantics does not oblige the patterns to be exhaustive on the type of E, it may be that no local transition at all can be derived from a case action (for a given ρ or in general).

Conditional action

For several imaginable applications, the **case** action is cumbersome e.g., when there are at most two alternatives and the conditions do not need pattern-matching to be checked. Thus, ATLANTIF also provides a simple conditional action.

$$\begin{array}{rl} A ::= & \cdots \\ | & \textbf{if } E \textbf{ then } A_1 \text{ [}\textbf{else } A_2\text{] } \textbf{end} \text{ [}\textbf{if}\text{]} \\ | & \cdots \end{array}$$

If the optional **else**-branch is unspecified, then we suppose it to be equivalent to "**else null**".

Static semantics. The **if**-action is well-bound if both A_1 and A_2 are well-bound. It is well-typed if $type(E) = \textbf{bool}$ and if both A_1 and A_2 are well-typed. The sets *read* and *write* are defined as follows:

$$\begin{aligned} read(\textbf{if } E \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ end}) &\stackrel{\text{def}}{=} read(E) \cup read(A_1) \cup read(A_2) \\ write(\textbf{if } E \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ end}) &\stackrel{\text{def}}{=} write(A_1) \cup write(A_2) \end{aligned}$$

Dynamic semantics. A_1 is executed, if E evaluates to **true**, and A_2 otherwise.

$$(if_1)\ \frac{eval(E, \rho, \textbf{true}) \wedge (A_1, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho')}{(\textbf{if } E \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ end}, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho')}$$

$$(if_2)\ \frac{eval(E, \rho, \textbf{false}) \wedge (A_2, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho')}{(\textbf{if } E \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ end}, d, \rho) \stackrel{l}{\Longrightarrow} (s, d', \rho')}$$

Note that the **if**-action is case-exhaustive, as the **else**-branch is always explicitly or implicitly declared.

Nondeterministic choice

$$\begin{array}{rl} A ::= & \cdots \\ | & \textbf{select } A_0 \,[]\, \ldots \,[]\, A_n \textbf{ end} \text{ [}\textbf{select}\text{]} \\ | & \cdots \end{array}$$

In the **select** action, any action A_i ($i \in 0..n$), may be executed.

Static semantics. The **select**-action is well-bound if for each $i \in 0..n$, A_i is well-bound. It is well-typed if for each $i \in 0..n$, A_i is well-typed. The sets *read* and *write* are defined as follows:

$$\begin{array}{l} \mathit{read}(\textbf{select}\ A_0\,[]\ \ldots\ []\,A_n\ \textbf{end}) \stackrel{\text{def}}{=} \bigcup_{i\in 0..n} \mathit{read}(A_i) \\ \mathit{write}(\textbf{select}\ A_0\,[]\ \ldots\ []\,A_n\ \textbf{end}) \stackrel{\text{def}}{=} \bigcup_{i\in 0..n} \mathit{write}(A_i) \end{array}$$

Dynamic semantics. Select actions define local transitions as follows:

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

Loop

$$\begin{array}{rcl} A ::= & & \cdots \\ & \mid & \textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}\ [\textbf{while}] \\ & \mid & \cdots \end{array}$$

This action repeatedly executes A_0, as long as the condition E is satisfied.

Static semantics. The **while**-action is well-bound if A_0 is well-bound. It is well-typed if $\mathit{type}(E) = \textbf{bool}$ and if A_0 is well-typed. The sets *read* and *write* are defined as follows:

$$\begin{array}{l} \mathit{read}(\textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}) \stackrel{\text{def}}{=} \mathit{read}(E) \cup \mathit{read}(A_0) \\ \mathit{write}(\textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}) \stackrel{\text{def}}{=} \mathit{write}(A_0) \end{array}$$

Dynamic semantics. While actions define local transitions as follows:

$$(\mathit{while}_1)\ \frac{\mathit{eval}(E, \rho, \textbf{true}) \land (A_0\,\textbf{;}\,\textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}, d, \rho) \stackrel{l}{\Longrightarrow} \sigma}{(\textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}, d, \rho) \stackrel{l}{\Longrightarrow} \sigma}$$

$$(\mathit{while}_2)\ \frac{\mathit{eval}(E, \rho, \textbf{false})}{(\textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end}, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, d, \rho)}$$

As the semantics rules enable only finite derivations, only finite loops actually have semantics.

Inaction

$$\begin{aligned} A ::= \;& \cdots \\ \mid \;& \mathbf{null} \end{aligned}$$

The **null** action simply executes; without changing variable values or the phase, and without invoking a communication gate.

Static semantics. A **null** action is always well-bound and well-typed. The sets *read* and *write* are defined as follows:

$$\mathit{read}(\mathbf{null}) \stackrel{\text{def}}{=} \varnothing \qquad \mathit{write}(\mathbf{null}) \stackrel{\text{def}}{=} \varnothing$$

Dynamic semantics. **null** actions define local transitions as follows:

$$(\mathit{null})\ \frac{}{(\mathbf{null}, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, d, \rho)}$$

Obviously, the **null** action is a neutral element for sequential composition.

4.4.3 Unit semantics

Static semantics. Discrete state names have to be pairwise distinct within one unit. Different units may contain states with the same name.

An *execution path* of a multibranch transition "**from** s A" is a succession of atomic actions (i.e., assignments, resets, delays, communications, jumps, stops, and **null**). For each multibranch transition, we demand the *unicity of communication and undelayed next state reachability*. The unicity of communication expresses that on each execution path, there is at most one communication action. The undelayed next state reachability expresses that each path containing a communication action ends with a jump to another state, without time being enabled to elapse between the communication and the jump. This constraint is necessary to ensure that, after a synchronization, a unit is always immediately in a new discrete state. Functions formally defining this criterion are given in Appendix A.1.1.

The sets $\mathit{read}(u)$ and $\mathit{write}(u)$ contain all variables that are respectively read and written in the multibranch transitions **from** s_0 A_0 ... **from** s_m A_m of the unit u. They are defined as follows:

$$\mathit{read}(u) \stackrel{\text{def}}{=} \bigcup_{i \in 0..m} \mathit{read}(A_i) \qquad \mathit{write}(u) \stackrel{\text{def}}{=} \bigcup_{i \in 0..m} \mathit{write}(A_i)$$

Dynamic semantics. The dynamic semantics of units is formally defined by a relation $(A, d, \rho) \overset{l}{\Longrightarrow} (s, d', \rho')$, which is the same notation as the semantics of actions (cf. Section 4.4.2). A transition corresponds to a succession of execution paths, the last of which is the only path to contain a gate communication. Transitions labelled ε have to be eliminated i.e., merged with a following transition, because only those transitions that are labelled by a gate communication allow synchronizations with other units. These successions are implemented by the rule (ε-*elim*) as follows:

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

Note that before, the rule (seq_1) introduced the elimination of ε-transitions in a single execution path. The same idea is applied here to a succession of execution paths by the rule (ε-*elim*).

4.4.4 Subunits

$$\begin{aligned} U ::= {} & \textbf{unit}\ u\ \textbf{is} \\ & \ldots \\ & U_1\ \ldots\ U_l \\ & \textbf{end unit} \end{aligned}$$

$U_1, \ldots, U_l$ are the subunits of u. They also may contain subunits themselves, thus defining a hierarchy of units.

Static semantics

Hierarchical order. The concept of subunits enables the definition of a partial order "$\succ$" on $\mathbb{U}$, by $u \succ u'$ if and only if u' is a direct or indirect subunit of u. We write $u \succeq u'$ for $u \succ u'$ or $u = u'$.

Variables defined in u can be read and/or written in the subunits of u, thus enabling sharing of variables between units. Therefore, variable identifiers have to be globally unique (we will detail this point in Section 4.6.2 and in Remark 4.15). We say that u is well-bound if and only if the multibranch transition actions $A_0, \ldots, A_n$ are well-bound and for each variable V in $read(u) \cup write(u)$, there exists a $u' \succeq u$ such that $V \in decl(u')$.

Variable scope. Each variable V has a scope, given by the set $accessible(V)$, that defines the units in which V can be read and/or written. Scopes ensure that a unit that may write V cannot be active at the same time as another unit that may read and/or write V (i.e., the sharing of V is limited by its scope). The variable will be called *well-accessible* if and only if such a scope can be defined.

For instance, let u_1 be a unit declaring a variable V_0 and let u_2, u_3 be subunits of u_1. If u_2 contains a multibranch transition "**from** s_1 V_0 `:=` 5; G; **to** s_2" and u_3 contains a multibranch transition "**from** s_3 V_0 `:= 3; ` G`;` **to** s_4", then a synchronization on G by these transitions would assign the values 5 and 3 to V_0 at the same time. Such ambiguities have therefore to be prohibited.

According to the idea that subunit hierarchy represents a process at different levels of granularity (cf. Section 4.2), it is an intuitive restriction to require that for two units u, u' such that $u \succ u'$, at most one of them may be active at the same time. This restriction, called *well-activatedness*, which will be detailed in Section 4.6.2, is the key element in the verification of whether a correct scope can be defined.

A variable V, declared in u, written in the units $u_1, \ldots, u_n$ (i.e., these are exactly those units such that $V \in \mathit{write}(u_i)$ with $i \in 1..n$), and read in the units $u'_1, \ldots, u'_m$ (i.e., exactly those units such that $V \in \mathit{read}(u_i)$ with $i \in 1..m$) is *well-accessible*, if it satisfies the following two constraints:

- To avoid write/write conflicts, in any moment at most one of the units $u_1, \ldots, u_n$ may be active. Supposing the module is well-activated, it is sufficient to require 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 $u'_i \notin \{u, u_1, \ldots, u_n\}$ is active in a given moment, no unit of $\{u, u_1, \ldots, u_n\}$ may be active at the same time. Supposing the module is well-activated, it is sufficient to require that the set $\{u'_i, u_0, u_1, \ldots, u_n\}$ is totally ordered by $\succ$.

An example of a well-accessible shared variable is shown in Fig. 4.2, where each circle represents one unit and the arcs represent a direct subunit relation; thus the figure shows the unit hierarchy of a module.

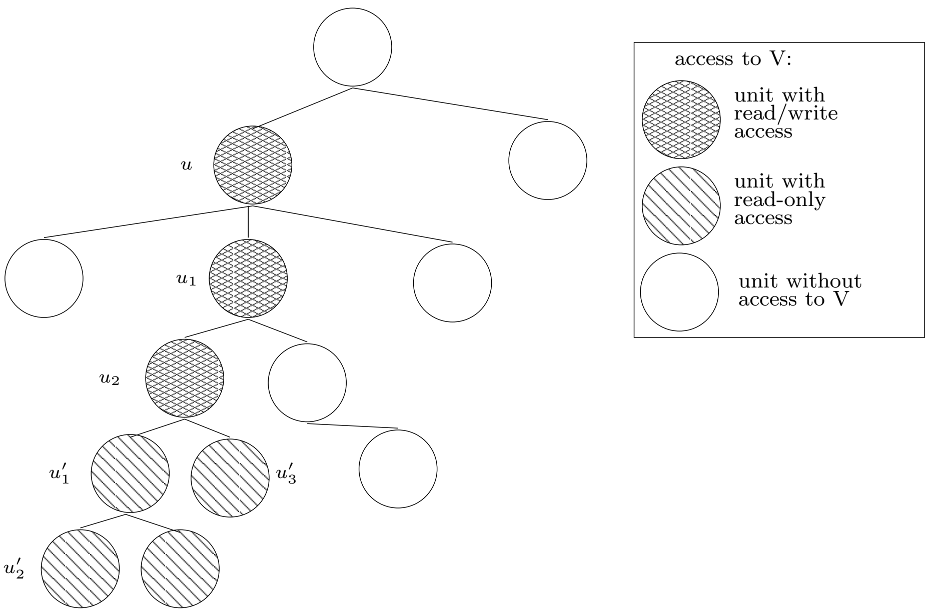

Figure 4.2: Example of a well-accessible shared variable

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 (using $u, u_1, \ldots, u_n$ as defined above):

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

Note that $\mathit{accessible}(V)$ can be a strict superset of the set of units in which V actually occurs. Intuitively it is the maximal set of units in which using V is allowed.

We use the notation $\rho_{\lceil\mathcal{U}}$ to restrict the domain of store ρ to those variables that are accessible in a unit set $\mathcal{U}$, formally $\rho_{\lceil\mathcal{U}} \stackrel{\text{def}}{=} \rho \ominus \{V \mid (\forall u \in \mathcal{U})\ u \notin \mathit{accessible}(V)\}$.

Example 4.2. *We suppose a module that models the control systems of a car, containing (among others) a unit* Brakes *with two subunits* Front_Brakes *and* Rear_Brakes.

When unit Brakes *prepares a braking operation, it writes a variable* Target_Speed, *which is read by its two subunits. Moreover,* Brakes *writes a variable* Front_RPM *that is also written in unit* Front_Brakes. *Symmetrically, there is a variable* Rear_RPM *written in* Brakes *and* Rear_Brakes. *Thus, the access-sets of these variables are as follows:*

$$\begin{aligned}
&\mathit{accessible}(\mathit{Target_Speed}) = \{\mathit{Brakes}, \mathit{Front_Brakes}, \mathit{Rear_Brakes}\}\\
&\mathit{accessible}(\mathit{Front_RPM}) = \{\mathit{Brakes}, \mathit{Front_Brakes}\}\\
&\mathit{accessible}(\mathit{Rear_RPM}) = \{\mathit{Brakes}, \mathit{Rear_Brakes}\}
\end{aligned}$$

4.5 Synchronizers

4.5.1 Syntax description

A synchronizer has the form

$$\textbf{sync}\ G[:B]\ \textbf{is}\ K\ [\textbf{stop}\ u_1\textbf{,}\ \ldots\textbf{,}u_m]\ [\textbf{start}\ u'_1\textbf{,}\ \ldots\textbf{,}u'_n]\ \textbf{end sync},$$

where:

- G is the name of the gate that triggers the synchronizer (by communication actions on G).
- B is an optional tag attached to G, written $\mathit{tag}(G)$, which may take one of four different values: **visible** induces a transition labelled by G and the offers exchanged on G; **hidden** induces an internal transition labelled by τ; **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*, defined after the following grammar:

$$
\begin{array}{rcll}
K & ::= & u & (\textit{single unit}) \\
 & | & K_1 \textbf{ and } K_2 & (\textit{synchronization}) \\
 & | & K_1 \textbf{ or } K_2 & (\textit{alternative}) \\
 & | & N \textbf{ among } (K_1, \ldots, K_m) & \\
 & | & (K_0) & \\
N & ::= & n & (\textit{natural integer}) \\
 & | & N_1 \textbf{ or } N_2 & (\textit{choice})
\end{array}
$$

- "**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 and $u'_1, \ldots, u'_n$ become active when the synchronizer is triggered. We write $stop(G) = \{u_1, \ldots, u_m\}$ and $start(G) = \{u'_1, \ldots, u'_n\}$. If the lists are unspecified, we assume $stop(G) = \varnothing$ and $start(G) = \varnothing$ respectively.

While avoiding an explosion of the number of transitions (cf. Section 3.5.1), synchronizers are 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 [64] can also be expressed by synchronizers. For instance, the composition "**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**". This feature will be detailed in Section 5.5.

- Processes can be started and 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.

4.5.2 Static semantics

Each synchronizer must have a unique gate. Therefore, a gate can be seen as a synchronizer's identifier. Furthermore, we require that each unit occurring in the formula K or in the lists of started and stopped units is declared within the same module. A synchronizer must not be tagged **urgent** if the module has the timing option **no time**.

Validity-stable synchronizers

In this section, we define the property of *validity-stability* for synchronizers, which intuitively assures that the starting of units by this synchronizer never leads to a state in which an active unit is a (direct or indirect) subunit of another active unit.

First, we define the predicate *valid_active* on $\mathcal{P}(\mathbb{U})$, which expresses that no unit in a set is a subunit of another unit in the same set. Formally:

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

When a synchronization on a gate G leads from a set of active units $\mathcal{U}$ with $valid\_active(\mathcal{U})$ to a set of active units $\mathcal{U}'$, it has to be guaranteed that $valid\_active(\mathcal{U}')$ is **true**. To this aim, it is sufficient to require the following:

- A unit that is in $\mathcal{U}$ can only be started by G if it is also stopped by G.
- The union of $\mathcal{U}$ and the units started by G, 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:

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

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. The predicate is defined by a quantification on the power set of $\mathbb{U}$, thus a naïve implementation would have an exponential complexity. In practice, an alternative but equivalent predicate can be implemented, which induces an algorithm of polynomial complexity. Moreover, if a synchronizer is not validity-stable, this algorithm automatically reports which units have to be stopped, in order to establish validity-stability. A description of this alternative definition, along with a formal proof of equivalence, can be found in Appendix A.1.2.

4.5.3 Dynamic semantics

The dynamic semantics of a synchronizer G defines a set of *synchronization sets*, written $sync(G)$. Each synchronization set is a set of units and represents the possibility of a synchronization[14] on G between all units of this set. Formally:

[14] If this set is a singleton, it is of course not entirely appropriate to speak of a "synchronization", which intuitively should include at least two units. However, for the sake of simplicity, we will use the term "synchronization" in the more general sense that also includes a communication performed by a single unit.

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

The predicate *sync* will be used in the module semantics defined in Section 4.6.3.

Remark 4.2. *Note that the semantics of the* **or** *does not correspond to the "or" of classical logic because e.g.,* $\{u_1, u_2\} \notin \mathit{sync}(u_1 \textbf{ or } u_2)$*, nor does it correspond to the exclusive or ("xor") because e.g.,* $\{u_1\} \in \mathit{sync}(u_1 \textbf{ or } u_1)$*. Instead, it corresponds to the* additive conjunction *"&" of linear logic [66], which represents one of the common usages of the word "or" in natural language, as for instance on the menu card of a restaurant, where the starter might be "soup* or *salad".*

Example 4.3. *We consider again the car brakes from Example 4.2, where* Brakes *corresponds to the brakes of a car, and* Front_Brakes *and* Rear_Brakes *are its subunits. Whenever the car is driving without braking, unit "*Brakes*" is active, and the two subunits otherwise. Such a behaviour is expressed by the following two synchronizers:*

$$
\begin{array}{l}
\textbf{sync } \mathit{Begin\_Braking} : \textbf{ silent is } \mathit{Brakes} \\
\qquad \textbf{stop } \mathit{Brakes} \textbf{ start } \mathit{Front\_Brakes}, \mathit{Rear\_Brakes} \textbf{ end sync} \\
\textbf{sync } \mathit{Finish\_Braking} : \textbf{ silent is } \mathit{Front\_Brakes} \textbf{ and } \mathit{Rear\_Brakes} \\
\qquad \textbf{stop } \mathit{Front\_Brakes}, \mathit{Rear\_Brakes} \textbf{ start } \mathit{Brakes} \textbf{ end sync}
\end{array}
$$

4.6 Modules

4.6.1 Syntax description

Modules are the top level construct of ATLANTIF and defined as follows:

$$
\begin{array}{rl}
X ::= & \textbf{module } M \textbf{ is} \\
& [(\textbf{no} \mid \textbf{discrete} \mid \textbf{dense}) \textbf{ time}] \\
& \textbf{type } T_1 \textbf{ is } D_1 \ldots \textbf{ type } T_n \textbf{ is } D_n \\
& \textbf{function } F_1 \textbf{ is } Y_1 \ldots \textbf{ function } F_k \textbf{ is } Y_k \\
& R_1 \ \ldots \ R_m \\
& \textbf{init } u_0, \ldots, u_j \\
& U_0 \ \ldots \ U_l \\
& \textbf{end module}
\end{array}
$$

This definition is composed of the following:

- M is the module's identifier.

- The optional *timing option* of the form **no time**, **discrete time**, or **dense time** indicates which time domain applies for this module. If no timing option is specified, it is taken to be **no time**.

  Note that **no time** merely indicates that time is abstracted from in the model i.e., it does not mean that no time elapses in the modeled system.

- $T_1, \ldots, T_n$ are type identifiers, $D_1, \ldots, D_n$ are type definitions, $F_1, \ldots, F_k$ are function identifiers, and $Y_1, \ldots, Y_k$ are function definitions (cf. Section 4.3.1).

- $R_1, \ldots, R_m$ are synchronizers, as defined in Section 4.5.

- The list $u_0, \ldots, u_j$ identifies the initially active units. We write the set $\mathcal{U}_0 \stackrel{\text{def}}{=} \{u_0, \ldots, u_j\}$.

- $U_0, \ldots, U_l$ are units (possibly containing additional subunits), as defined in Section 4.4.

4.6.2 Static semantics

Unique identifiers. Each of the following sets of constructs must have a separate name space i.e., it must not contain two constructs that have the same identifier:

- types
- constructors and functions
- units (including subunits)
- synchronizers
- variables

Additionally, syntax keywords as well as identifiers of predefined types and functions may not be chosen as identifiers in general. Appendix C on page 257 contains a full list of these reserved keywords.

The elements of the list of initial units must be pairwise distinct.

Well-declaredness. All initial units must be declared i.e., $\mathcal{U}_0 \subseteq \mathbb{U}$. Also, all functions, all synchronizers, and all units have to be well-declared.

Well-initialized variables. The module must be *well-initialized* i.e., each time a variable is read, it has been written previously and not reset since the last writing. An algorithm implementing a sufficient criterion for a single unit is presented in [61], but it does not consider variable sharing or starting and stopping of units, both concepts provided by ATLANTIF. Therefore, ATLANTIF uses a new algorithm, which is described in Appendix A.1.3. In short, it creates a finite directed graph where each node corresponds to a set of states of the ATLANTIF semantics and each edge to a set of possible transitions between these states, which has three variable sets associated:

- the variables that are written during this transition
- the variables read during this transition but not being defined earlier on the same transition
- the variables that are reset during this transition

Given this graph, a minimal fixpoint of the variables defined in each node is calculated. Then, the module is well-initialized if for each edge, the second variable set is a subset of the minimal fixpoint set of its source node.

Well-activatedness. An ATLANTIF module must be *well-activated* i.e., at any moment of an execution, the set of active units cannot contain two units such that one is a (direct or indirect) subunit of another. This follows the idea that subunits represent the same system at a finer granularity.

Well-activatedness is satisfied if both the following are true:

- its set of initially active units $\mathcal{U}_0$ satisfies $valid\_active(\mathcal{U}_0)$ (cf. Section 4.5.2)
- each synchronizer G satisfies $validity\_stable(G)$ (cf. Section 4.5.2)

Correct usage of silent and urgent synchronizers. Time windows control the time elapsing between two discrete (i.e., **visible**, **hidden**, or **urgent**) synchronizations. Thus, if G is a **silent** synchronizer, communication actions using G must not have a time window. Instead, the silent synchronization is understood to happen as early as possible. Note that time can elapse before a silent synchronization, if a **wait** action occurs.

If G is an **urgent** synchronizer, then each interval occurring in the time window W of a communication action using G must be left-closed. This restriction is necessary to avoid timelocks: Supposing a synchronizer "**sync** G : **urgent is** u **end sync**" and the unit u containing a communication action "G **in**] x, . . . [" in a module with a dense time domain. Then, when x time units have elapsed, a communication by G is still not possible, because x is below the time window; but a further time elapsing by any t time units is not possible either, because an urgent communication would have been possible earlier (e.g., after $t/2$ time units).

4.6.3 Dynamic semantics

Modules being the top-level constructs of ATLANTIF, their dynamic semantics are the semantics of an entire system modelled in ATLANTIF. Thus, this section will "assemble" all semantics definitions from the above section into the rules defining a timed labelled transition system from an ATLANTIF module.

Semantic model. The time domains we consider in the semantics definition are $\{0\}$ if the timing option is **no time**, $\mathbb{N}$ if the timing option is **discrete time**, and $\mathbb{R}_{\geq 0}$ if the timing option is **dense time** (cf. Section 3.1.1 for a discussion on the differences between $\mathbb{R}_{\geq 0}$ and $\mathbb{Q}_{\geq 0}$). Indicating a "time domain" for specifications tagged as untimed might seem strange, but this will allow us to have only one definition for (timed) semantics, in which untimed behaviour can be seen as a special case (cf. Remark 4.3). Like in Definition 3.2, we write $\mathbb{D}$ for the time domain.

Note that our choice for a time domain will not affect the semantics rules.

The dynamic semantics of an ATLANTIF specification is given by a TLTS (cf. Definition 3.3) of the form $(\mathbb{S}, \mathbb{L}'_1, \mathbb{D}, \mathbb{T}, S_0)$, where:

- $\mathbb{S}$ is a set of *global states* (or sometimes simply *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 u to its current discrete state in $\mathcal{S}_u$
  - $\theta : \mathbb{U} \to (\mathbb{D} \times \mathbf{Bool})$ is a partial function, called *time distribution*, that maps each active unit to its current real-time status (phase and blocking condition), as described in the unit semantics (cf. Section 4.4.2)
  - ρ is a *store* i.e., a partial function that maps the currently assigned variables to their values (cf. Section 4.3.2)

  Note that the set of active units is given by $dom(\pi)$ and $dom(\theta)$, with $dom(\pi) = dom(\theta)$.

  When time elapses and no gate communication is performed, we do not consider what else happens in the unit (which is a black box). For this reason, we represent a unit in the module semantics basically by two pieces of information, which are the state of the unit directly after its last gate communication (provided by the function π), and the amount of time that elapsed since then (the phase, cf. Section 4.4.2).

- $\mathbb{L}'_1 \stackrel{\text{def}}{=} (\mathbb{L}_1 \setminus \{\varepsilon\}) \cup \{\tau\}$ is the set of discrete labels. It contains gates with values as well as the label τ, which represents discrete transitions by hidden gates.

- $\mathbb{T}$ is a set of transitions defined as a relation in $\mathbb{S} \times \mathbb{L}_2 \times \mathbb{S}$, where $\mathbb{L}_2 \stackrel{\text{def}}{=} \mathbb{L}'_1 \cup (\mathbb{D} \setminus \{0\})$.

- $S_0 \in \mathbb{S}$ is the initial state, which is defined by $S_0 \stackrel{\text{def}}{=} (\pi_0 \ominus \bar{\mathcal{U}_0}, \theta_0 \ominus \bar{\mathcal{U}_0}, \rho_{0\,|\mathcal{U}_0})$, 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 $(0, \mathbf{false})$ for each unit.
- ρ_0 is the store that maps each variable to its initial value, if any.
- $\bar{\mathcal{U}_0} \stackrel{\text{def}}{=} \mathbb{U} \setminus \mathcal{U}_0$ i.e., $\bar{\mathcal{U}_0}$ is the complementary set of $\mathcal{U}_0$. Thus, $\pi_0 \ominus \bar{\mathcal{U}_0}, \theta_0 \ominus \bar{\mathcal{U}_0}$ denote respectively the restriction of π_0 and θ_0 to the domain $\mathcal{U}_0$.

$\rho_{0\lceil\mathcal{U}_0}$ denotes the restriction of a store to the variables accessible in $\mathcal{U}_0$ i.e., $\rho_{0\lceil\mathcal{U}_0} = \rho_0 \ominus \{V \mid (\forall u \in \mathcal{U}_0)\ u \notin \mathit{accessible}(V)\}$, as formally defined in Section 4.4.4.

Chains of synchronizations. As we indicated in the definition of synchronizers (cf. Section 4.5.1), only synchronizations by visible, hidden, and urgent synchronizers induce discrete transitions. Thus, a discrete transition corresponds to a chain of zero or more silent synchronizations, followed by a non-silent synchronization (hereafter called a *chain*). A chain executes without time elapsing.

As observed in [59] for a very similar context, it would be incorrect to explore all possible chains. For instance, suppose that at a given moment two units u_1, u_2 could either synchronize on a silent synchronizer G_1 or on a visible synchronizer G_2 (both excluding the other possibility), and at the same time, two units u_3, u_4 could synchronize on a visible synchronizer G_3 (cf. Fig. 4.3).

Figure 4.3: Independent synchronizations

Then, a chain only consisting in a synchronization on G_3 and a chain consisting in a synchronization on G_1, followed by a synchronization on G_3 would both seem equal to an external observer. However, in the first case, a synchronization on G_2 would be possible afterwards, but not in the second case, which is clearly not intuitive: When the only observable event (G_3) occurred independently of the units u_1 and u_2, then nothing should change within u_1 and u_2. Therefore, there should be a restriction to those chains, where each silent synchronization is directly or indirectly necessary for the non-silent synchronization. More precisely, we demand that 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. The idea behind these two conditions will be further illustrated by the Examples 4.4 to 4.6 in Section 4.7.1.

To formally describe the chains that should be explored, we define the *unit affection* of a synchronization as the set containing those units one of which must be relevant for another synchronization in the remainder of the chain i.e., the set consisting of the synchronizing and not stopped units and of the started units. We call an *incomplete chain* any strict prefix of a chain and we construct for 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.

Predicates used in the module semantics. In the following, we define several predicates that will be used in the TLTS rules.

The predicate *synchronizing*$((S,\alpha),l,\mu,(S',\alpha'))$, defined on $(\mathbb{S}\times\mathcal{P}(\mathcal{P}(\mathbb{U})))\times(\mathbb{L}'_1\setminus\{\tau\})\times$ **Bool** $\times(\mathbb{S}\times\mathcal{P}(\mathcal{P}(\mathbb{U})))$, is **true** if and only if the following are all true:

- A transition labelled l may occur in global state S and leads to global state S'.
- The disjunction of the blocking conditions in the local states reached via this transition equals μ.
- A set of unit affections α evolves to α' via this synchronization.

Formally:

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

where *next_π*, *next_θ*, *next_ρ*, and *next_α* are defined below.

Note that $\rho_i(V)$ in the second argument of the predicate *next_ρ* is indeed well-defined (cf. Remark 4.6). Note also that this definition requires the offers "$v_1\ldots v_n$" to be the same in all local transitions i.e., they need to match in number, types, and values.

Intuitively, the predicate *synchronizing* corresponds to a possible synchronization that leads from one global state (extended with a unit affection) to another. It "picks up" transitions of the action semantics (relation "$\Longrightarrow$", cf. Section 4.4.2) and combines them following the semantics of the applying synchronizer (set *sync*(G), cf. Section 4.5.3). Only those local transitions that correspond to execution paths that end with a jump action or a **stop** action are picked up; other paths being ignored.

Moreover, *synchronizing* uses the two mappings *act* (providing the action associated with a unit's discrete state) and *accessible* (providing the set of units in which a variable can be read and/or written), which we formally defined in Sections 4.4.1 and 4.4.4 respectively.

The predicates $next\_\pi$, $next\_\theta$, $next\_\rho$, and $next\_\alpha$ are defined as follows:

- The predicate $next\_\pi(\pi, \pi_1, G, \pi')$ defines the new state distribution after a synchronization. The argument π corresponds to the state distribution before the synchronization, π_1 to the state distribution in the synchronizing units after the synchronization, G to the synchronizer (determining unit stopping and starting by the sets $stop(G)$ and $start(G)$), and π' to the new state distribution.

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

- The predicate $next\_\theta(\theta, \mathcal{U}, t_0, G, \theta')$ defines the new time distribution after a synchronization. The arguments θ and θ' correspond to the old and the new time distribution respectively, $\mathcal{U}$ corresponds to the set of synchronizing units, and G to the applying synchronizer.

  The synchronizer is needed not only to determine the domain of θ', but also to determine the new values, using its visibility $tag(G)$: If the synchronization was **silent**, the new phase t_0 of the affected units is given by the minimum of the phases 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 (which will be further illustrated in Example 4.6). Otherwise, the new phase of the affected units is set to 0.

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

- The predicate $next\_\rho(\rho, \rho_1, \mathcal{U}, G, \rho')$ defines the new store after a synchronization. The arguments ρ and ρ' correspond to the old and the new store respectively, ρ_1 corresponds to the store that maps each variable accessible in the synchronizing units to its value after the synchronization, $\mathcal{U}$ corresponds to the set of units that are active after the synchronization, and G to the applying synchronizer.

  The domain of ρ' is restricted according to $\mathcal{U}$. For each started unit declaring a variable V with an initial value (given by the set $decl(u)$), the new store is extended by $[V \mapsto \rho_0(V)]$.

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

- The predicate $\mathit{next\_\alpha}(\alpha, \mathcal{U}, G, \alpha')$ formally defines the new unit affections after a synchronization, as described above. The arguments α and α' correspond to the old and the new unit affections respectively, $\mathcal{U}$ corresponds to the set of synchronizing units, and G is the applying synchronizer. To derive α' from α, first those unit affections from which at least one unit has synchronized are deleted, and then, if the synchronization was **silent**, a new set is added, containing all synchronizing and not stopped units and all started units.

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

The predicate $\mathit{enabled}(S, l, \mu, S')$, defined on $\mathbb{S} \times (\mathbb{L}_1' \setminus \{\tau\}) \times \textbf{Bool} \times \mathbb{S}$, is **true** if and only if there is a chain that satisfies the restrictions on silent synchronizers described above, starting in global state S and ending in global state S', whose visible synchronization is labelled l and where the blocking condition equals μ. Formally:

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

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 with a strict deadline that has been reached (i.e., $\mu = \textbf{true}$). The predicate $\mathit{relaxed}(S)$, defined on $\mathbb{S}$, is **true** if and only if time can elapse in S. Formally:

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

Discrete and timed transitions. Discrete transitions are defined by rule (*rdv*) as follows:

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

where function *label* transforms a label of $\mathbb{L}_1' \setminus \{\tau\}$ into a discrete label of $\mathbb{L}_2$:

$$label(G\ v_1 \ldots v_n) \stackrel{\text{def}}{=} \begin{cases} G\ v_1 \ldots v_n & \text{if } tag(G) = \textbf{visible} \\ \tau & \text{otherwise} \end{cases}$$

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 $\theta(u) = (t_u, \mu_u)$.

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

With this semantic approach, we respect the standard property that time must elapse at the same speed in all units, implemented by the definition of the operation "$\theta + t$".

Remark 4.3. *At the beginning of this section, we stated that untimed* ATLANTIF *specifications use the same semantic rules as the timed ones. This is possible, because rule* (time) *requires that $t > 0$. As the timing option "***no time***" induces $\mathbb{D} = \{0\}$, obviously rule* (time) *can never apply. Therefore, the untimed case can be presented as a subcase of the general (timed) semantics.*

Furthermore, as the following section shows, this semantics has the suitable properties mentioned in Section 3.1.1.

4.7 Properties of the semantics

4.7.1 Examples and remarks on the formal definitions

Synchronization chains. The three following examples will illustrate the idea behind the conditions we imposed on synchronization chains. Furthermore, the third one will also illustrate the application of the predicate *next_θ*.

Example 4.4. *The module of Fig. 4.4 describes an industrial machine with three subsystems for error-detection, each one for an error A, B, and C respectively. Error A is not problematic unless it happens too often (normal discard), error B is not problematic for itself, but is a possible cause for error C, where the latter can cause critical situations that damage the machine. Therefore, in a case of C, we have to analyze how many Bs occurred.*

In the module, one unit is defined for each error detection subsystem, and one silent synchronizer for each of the three error occurrences. In unit Detect_A, each communication on A_occurs increases the variable Count_A by 1; when Count_A reaches or passes beyond 100, a communication by Too_many_A is made. In unit Detect_B, each communication on B_occurs increases the variable Count_B by 1; at any time there can also be a communication on gate After_C, where the value of Count_B is transmitted by an offer. In unit

```
module Machine_Error_Detection is no time
sync A_occurs:silent is Detect_A end sync
sync B_occurs:silent is Detect_B end sync
sync C_occurs:silent is Detect_C end sync
sync Too_many_A is Detect_A end sync
sync After_C is Detect_B and Detect_C end sync

init Detect_A, Detect_B, Detect_C

unit Detect_A is
  variables Count_A : int := 0
  from AS1
    A_occurs; Count_A:=Count_A + 1;
    if (Count_A < 100) then to AS1
      else to AS2 end
  from AS2
    Too_many_A; to AS1
end unit

unit Detect_B is
  variables Count_B : int := 0
  from BS1
    select B_occurs;
            Count_B:=Count_B + 1;
            to BS1
          [] After_C !Count_B;
            to BS1 end select
end unit

unit Detect_C is
  from CS1
    C_occurs; to CS2
  from CS2
    After_C ?any int; to CS1
end unit

end module
```

Figure 4.4: Machine with error detections, illustrating chains

Detect_C, a communication on C_occurs is followed by a synchronization with Detect_B using After_C.

Different chains ending with After_C are possible, each of which contains arbitrarily many synchronizations on A_occurs and on B_occurs and one synchronization on C_occurs in any possible order. Thus, C_occurs must necessary happen before After_C, and the occurrences of B_occurs determine the offer of the communication on After_C (cf. the highlighted assignment in unit Detect_B). At the same time, occurrences of A_occurs influence neither the possibility nor the offers of a communication on After_C.

Therefore, chains ending with After_C should not contain A_occurs (which only concerns unit Detect_A), but these chains may contain B_occurs and C_occurs, which are the communications situated in those units performing the synchronization on After_C.

Example 4.5. *This example considers the influence of starting and stopping to chains. We suppose a module (given in Fig. 4.5) describing very schematically a system consisting of one subsystem (unit Main_System) performing repeatedly some task (synchronizer Job), and another subsystem (unit Activator), which is the only unit active in the beginning and that can perform the silent communication Activate that starts Main_System and stops Activator.*

Clearly, in the beginning only one chain is possible, which is a silent communication on Activate followed by a communication on Job. Although Activate comes from a different unit than Job, it nevertheless must necessarily occur before it, otherwise Main_Unit would not be active. More generally, the chains we should explore are not restricted to silent synchronizations with those units performing the non-silent synchronization, but also include silent synchronizations that start units in which communications occur later in the chain.

```
module External_Activation_1 is no time
sync Activate:silent is Activator stop Activator
        start Main_Unit end sync
sync Job is Main_Unit end sync

init Activator

unit Activator is
  from Launch
    Activate; stop
end unit

unit Main_Unit is
  from Ready
    Job; to Ready
end unit

end module
```

Figure 4.5: Subsystem started by an auxiliary subsystem (chain with starting)

Example 4.6. *This example considers chains with timed constructs and the developement of their phases. It extends the former example into a timed system, where we now suppose that two subsystems (units Activator_1 and Activator_2) are necessary to start the Main_System. One needs 3 seconds to prepare this activation, the other needs 5 seconds. After the activation, as before, the Main_System can repeatedly perfom the task "Job".*

```
module External_Activation_2 is discrete time
sync Activate:silent is Activator_1 and Activator_2
     stop Activator_1, Activator_2
     start Main_System end sync
sync Job is Main_System end sync

init Activator_1, Activator_2

unit Activator_1 is
  from Launch
    wait 3; Activate; stop
end unit

unit Activator_2 is
  from Launch
    wait 5; Activate; stop
end unit

unit Main_System is
  from Ready
    Job; to Ready
end unit
end module
```

Figure 4.6: Subsystem started by two auxiliary subsystems (chain with real time)

Again, only one chain is possible in the beginning, containing a silent communication on Activate followed by a communication on Job. Obviously, at least five seconds have to elapse before the synchronization on Activate, because until then, the unit Activator_2 would not be ready to participate. Thus, these five seconds are relevant for the calculation of the phase of Main_System when it is started. Note that the rule (comm) (cf. page 67) does not change the phase.

Example 4.7. Application of the rules. *We illustrate the semantics by deriving two* TLTS *transitions for the light switch shown in Example 4.1, page 55. We show that when*

User *is in state* Prepare_Work *with phase zero and* Lamp *in state* Low *also with phase zero, three time units may elapse before the button is pushed. Formally:* $(\pi, \theta, \varnothing) \xrightarrow{3} (\pi, \theta + 3, \varnothing) \xrightarrow{Push} ([User \mapsto Working, Lamp \mapsto Bright], \theta, \varnothing)$*, where* $\pi \stackrel{\text{def}}{=} [User \mapsto Prepare\_Work, Lamp \mapsto Low]$*, and* $\theta \stackrel{\text{def}}{=} [User \mapsto (0, \mathbf{f}), Lamp \mapsto (0, \mathbf{f})]$ *(where* $\mathbf{f}$ *is a shorthand for* **false***; similarly we will write* $\mathbf{t}$ *for* **true***).*

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

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

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

$$\frac{\begin{array}{c}\{User, Lamp\} \in \mathit{sync}(Push) \wedge (\mathit{act}(Prepare\_Work), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Working, (2, \mathbf{t}), \varnothing) \wedge \\ (\mathit{act}(Low), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Bright, (3, \mathbf{f}), \varnothing)\end{array}}{(\pi, \theta + 3, \varnothing) \xrightarrow{Push} ([User \mapsto Working, Lamp \mapsto Bright], \theta, \varnothing)}(\mathit{rdv})$$

The premise $(\mathit{act}(Prepare\_Work), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Working, (2, \mathbf{t}), \varnothing)$ *comes from the following, recalling that* $\mathit{act}(Prepare\_Work) = \textbf{wait}\ 1;\ Push\ \textbf{must in}\ \texttt{[0,2]}\ ;\ \textbf{to}\ Working$*:*

$$\frac{\dfrac{\mathit{eval}(1, \varnothing, 1) \wedge 3 \geq 1}{(\textbf{wait}\ 1, (3, \mathbf{f}), \varnothing) \stackrel{\varepsilon}{\Longrightarrow} (\delta, (2, \mathbf{f}), \varnothing)}(\mathit{wait}) \quad (Push\ \textbf{must in}\ \texttt{[0,2]}\ ;\textbf{to}\ Working, (2, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Working, (2, \mathbf{t}), \varnothing)}{(\mathit{act}(Prepare\_Work), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Working, (2, \mathbf{t}), \varnothing)}(\mathit{seq}_1)$$

Finally, the premise $(Push\ \textbf{must in}\ \texttt{[0,2]}\ ;\textbf{to}\ Working, (2, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Working, (2, \mathbf{t}), \varnothing)$ *comes from:*

$$\frac{\dfrac{\mathit{win\_eval}(\texttt{[0,2]}, \varnothing, [0,2]) \wedge 2 \in [0,2]}{(Push\ \textbf{must in}\ \texttt{[0,2]}, (2, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (\delta, (2, \mathbf{t}), \varnothing)}(\mathit{comm}) \quad \dfrac{}{(\textbf{to}\ Working, (2, \mathbf{t}), \varnothing) \stackrel{\varepsilon}{\Longrightarrow} (Working, (2, \mathbf{t}), \varnothing)}(\mathit{to})}{(Push\ \textbf{must in}\ \texttt{[0,2]}\ ;\textbf{to}\ Working, (2, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Working, (2, \mathbf{t}), \varnothing)}(\mathit{seq}_1)$$

*The derivation by rule (*comm*) calculates* $\mathit{up\_lim}(\textbf{must}, [0, 2], 2) = \textbf{true}$.

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

The two following remarks further illustrate the concept of unit affections in chains.

Remark 4.4. *"Dead-end" silent synchronizers are dead code:*

Supposing a specification contains a synchronizer of the form

"**sync** H**:silent is** *U1* **stop** *U1* **end sync***".*

Any chain prefix containing a synchronization by H *causes the unit affection set* α *to contain the set* $(\{U1\} \setminus \mathit{stop}(H)) \cup \mathit{start}(H) = \varnothing$*. Recalling that a continuation of a* *chain prefix can only delete sets in* α *if they contain certain units, an empty set can thus never be deleted. Therefore, the predicate* enabled *cannot apply to a chain containing a synchronization by* H*, just as we intuitively expect it.*

Note that it is not impossible to terminate a unit silently: As long as the termination includes the synchronization with a unit that is not stopped and/or another unit is started, silent synchronizers can be used to stop units.

Remark 4.5. *Bounds on sets of unit affections:*

A unit affection set α *is a subset of* $\mathcal{P}(\mathbb{U})$*, but it cannot be as big. First, the elements of* α *can only be such* $\mathcal{U}$ *for which* $valid\_active(\mathcal{U})$ *holds; this is a consequence of each synchronizer satisfying* validity-stability*. Second, the elements of* α *are necessarily disjoint. This can easily be derived from the definition of the predicate* $next\_\alpha$*.*

From the second observation, we can conclude that the upper bound for the cardinality of α *is* $card(\mathbb{U}) + 1$ *(instead of* $2^{card(\mathbb{U})}$*).*

Remark 4.6. *We show that the predicate* synchronizing *is indeed well-defined.*

The formal definition contains the term

$$next\_\rho(\rho, [V \mapsto \rho_i(V) \mid u_i \in accessible(V)], dom(\pi'), G, \rho'),$$

whose second argument could seem to be ambiguous: The argument $[V \mapsto \rho_i(V) \mid u_i \in accessible(V)]$ *is a function that maps each variable accessible in the synchronizing units* $u_1, \ldots, u_m$ *to its value after the synchronization. It is possible that there are* $j, k \in 1..m$ *with* $j \neq k$ *and a variable* V *such that the set* $accessible(V)$ *contains both* u_j *and* u_k*. The possible ambiguity lies in this case i.e., in the simultaneous application of* $V \mapsto \rho_j(V)$ *and* $V \mapsto \rho_k(V)$*.*

But in fact, we know by $u_j, u_k \in sync(G)$ *and* $validity\_stable(G)$ *that* $u_j \not\preceq u_k$ *and* $u_k \not\preceq u_j$*, and thus, with* V *being well-accessible, we can deduce that* V *is not defined (i.e., assigned a value) in* u_j *or* u_k *i.e.,* $\rho_j(V) = \rho_k(V)$*. Therefore, there is no ambiguity, and the predicate* synchronizing *is well-defined.*

Note that the same argument could be used to shorten the definition of the predicate synchronizing*: In the third line, the store is restricted to the term* $\rho_{\restriction\{u_i\}}$*. Keeping* ρ *instead of* $\rho_{\restriction\{u_i\}}$ *would not alter the semantics, but the predicate's intuition benefits from keeping the store restriction.*

4.7.2 Properties of the generated TLTS

Additivity, determinism, and maximal progress

Proposition 4.1. A TLTS constructed by the ATLANTIF formal semantics satisfies *time additivity*:

$$(\forall S_1, S_2 \in \mathbb{S}, t_1, t_2 \in (\mathbb{D} \setminus \{0\}))\ S_1 \xrightarrow{t_1+t_2} S_2 \text{ iff } (\exists S_3 \in \mathbb{S})\ S_1 \xrightarrow{t_1} S_3 \xrightarrow{t_2} S_2$$

(cf. Definition 3.4 *(i)*).

Proof. Let S_1 be a global state and $t_1, t_2 \in (\mathbb{D} \setminus \{0\})$. We define $S_1 \stackrel{\text{def}}{=} (\pi, \theta, \rho)$. 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:

$$\begin{array}{rl} & (\pi,\theta,\rho) \xrightarrow{t_1+t_2} (\pi, \theta + (t_1+t_2), \rho) \\ \text{iff} & (\pi,\theta,\rho) \xrightarrow{t_1} (\pi, \theta + t_1, \rho) \text{ and } (\pi, \theta+t_1, \rho) \xrightarrow{t_2} (\pi, (\theta+t_1)+t_2, \rho) \end{array}$$

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

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

□

Proposition 4.2. A TLTS constructed by the ATLANTIF formal semantics satisfies *time determinism*:

$$(\forall S_1, S_2, S_3 \in \mathbb{S}, t \in (\mathbb{D} \setminus \{0\})) \text{ if } S_1 \xrightarrow{t} S_2 \text{ and } S_1 \xrightarrow{t} S_3, \text{ then } S_2 = S_3$$

(cf. Definition 3.4 *(ii)*).

Proof. Let S_1, S_2, S_3 be global states, and $t \in (\mathbb{D} \setminus \{0\})$, such that $S_1 \xrightarrow{t} S_2$ and $S_1 \xrightarrow{t} S_3$. Let $S_1 = (\pi_1, \theta_1, \rho_1)$.

As in the proof of Proposition 4.1, only rule (*time*) allows one to derive the timed transitions $S_1 \xrightarrow{t} S_2$ and $S_1 \xrightarrow{t} S_3$. The "+" in (*time*) being well-defined, we thus have $S_2 = (\pi_1, \theta_1 + t, \rho_1) = S_3$. □

Proposition 4.3. A TLTS constructed by the ATLANTIF formal semantics satisfies *maximal progress of urgent actions*, where the set U of urgent labels (cf. Definition 3.4 *(iii)*) consists of those labels representing discrete synchronizations on **urgent** synchronizers.

Formally, let S_1 be a global state allowing an urgent synchronization i.e., there exists a synchronizer G declared **urgent**, and $v_1, \ldots, v_n, \mu, S_2$ such that $\mathit{enabled}(S_1, G\ v_1 \ldots v_n, \mu, S_2)$. We state that for $t \in (\mathbb{D} \setminus \{0\})$, there is no S_3 such that $S_1 \xrightarrow{t} S_3$.

Proof. Let $t \in (\mathbb{D} \setminus \{0\})$, and suppose that there is an S_3 such that $S_1 \xrightarrow{t} S_3$.

Such a transition could only be derived from rule (*time*), thus the predicate $\mathit{relaxed}((\pi, \theta + 0, \rho))$ (with $S_1 = (\pi, \theta, \rho)$) needs to be satisfied. By definition of *relaxed*, this is in direct contradiction with the hypothesis and thus not possible; therefore time cannot elapse in S_1. □

Remark 4.7. *In spite of the difficulties stated in Remark 3.2, the proofs of Propositions 4.1, 4.2, and 4.3 are relatively simple, which has two reasons. First, only the rule* (time) *describes timed transitions, whereas other languages e.g.,* E-LOTOS, *have several rules for time elapsing. Second, the modus operandi of the rule* (time) *is purely symbolic: It represents an elapsing of t time units in the module by increasing the phase for all active* *units by t i.e., by a simple shift of t in the time distribution θ (instead of, for instance, calculating for each active unit its state t time units in the future). By this intuition, it is clear why time additivity and time determinism are proven so easily. We recall that a unit's phase expresses how much time has elapsed since the last visible communication of this unit.*

This (to our knowledge original) approach is inspired from the time determinism property itself: As we explained in Remark 3.2, a semantics satisfying time determinism has to represent the successor state of a timed transition by **all** *possible evolutions during the elapsed time. One possibility to do this would be a set containing all correponding states, but if nondeterministic constructs are evaluated, such a set might become large or even infinite. Therefore, it makes sense to only have a single state that represents all evolutions symbolically.* ATLANTIF *expresses this by the phase.*

Non zenoness

The ATLANTIF semantics do not satisfy non zenoness as presented in Definition 3.5, as the following example shows:

Example 4.8. *Consider the module of Fig. 4.7:*

module *Zeno_Behaviour* **is dense time**
sync *G* : **urgent is** *U1* **end sync**
init *U1*
unit *U1* **is**
  from *S1*
    G ; **to** *S1*
end unit
end module

Figure 4.7: An ATLANTIF specification describing the simplest form of zeno behaviour

This specification corresponds to a TLTS *with a single state and a single discrete transition labelled G. Obviously, in such a* TLTS*, only one run is possible, and that run is not diverging.*

To avoid the kind of zeno behaviour shown in the previous example (called "*spin behaviour*" in [90]), the semantics must somehow prohibit infinite successions of discrete transitions: Either it could allow *unbounded* finite successions of discrete transitions, or it could fix a maximal number of consecutive discrete transitions (e.g., one, if a duration is associated with each event like in [108]). Neither of these approaches is intuitive, because both question the independence of timed and discrete transitions, which we suppose by choosing TLTS as a semantics model. Furthermore, rules that prohibit zeno behaviour completely are likely to be more complex than the rules we gave for ATLANTIF.

Thus, it seems to make more sense to accept the possibility of zeno behaviour, and to leave it to formal verification to detect it. Nevertheless, if we cannot exclude cases like in Example 4.8, we can still state the following:

Proposition 4.4. *(i)* The Tlts generated from a discrete time Atlantif specification cannot contain a timelock state (i.e., in every state, either time elapsing or a discrete action is possible).

(ii) The Tlts generated from a dense time Atlantif specification where all timed expressions are constant values cannot contain a timelock state. (Recall that "timed expressions" are those occurring as parameters of **wait** actions and time windows.)

(iii) The Tlts generated from a dense time Atlantif specification where all timed expressions are constant values cannot be strict zeno (cf. Definition 3.6 on page 19).

Proof. *(i)* Suppose S to be a state in the Tlts of a discrete time Atlantif module M. It has to be shown that S is no timelock state.

Case 1: There is a discrete label l, a blocking condition μ, and a state S' such that $\mathit{enabled}(S, l, \mu, S')$. Then, by rule (rdv), we have $S \xrightarrow{l} S'$, thus S is not a timelock state.

Case 2: There are no l, μ, S' as in the above case. Thus, rule (rdv) cannot apply, and therefore we have to show that rule (time) can apply. Thus, we have to show that there is a $t > 0$ such that $(\forall t' < t)\ \mathit{relaxed}((\pi, \theta + t', \rho))$ (where we suppose $S = (\pi, \theta, \rho)$).

By case hypothesis, we know that $\mathit{relaxed}((\pi, \theta + 0, \rho))$, because the definition of this predicate contains as a condition the existence of such l, μ, S' (cf. page 85). Therefore, as the time domain is discrete, the statement is satisfied by $t = 1$.

(ii) Suppose S to be a state in the Tlts of a dense time Atlantif module M such that all timed expressions are constants. It has to be shown that S is not a timelock state.

There is only a finite number of **wait** actions in M, we note their expressions (which are constants by hypothesis) $c_1, \ldots, c_n$. There is a finite number of non-silent synchronizers in M, and for each G among them there is a finite number of occurrences; we note the minima and the infima of the according time windows (which are also constants by hypothesis) $c_1^G, \ldots, c_m^G$. Then, it is possible to define a set Δ_G as follows:

$$\Delta_G \stackrel{\text{def}}{=} \{x_1 \cdot c_1 + \ldots + x_n \cdot c_n + c_i^G \mid i \in 1..m, x_1, \ldots, x_n \in \mathbb{N}\}$$

Moreover, it is possible to define a subset $\tilde{\Delta_G} \subseteq \Delta_G$ as follows:

$$\tilde{\Delta_G} \stackrel{\text{def}}{=} \{x_1 \cdot c_1 + \ldots + x_n \cdot c_n + c_i^G \mid i \in 1..m, x_1, \ldots, x_n \in \mathbb{N}, \text{ there is a path from } S \text{ to a communication on } G, \text{ corresponding to } x_1, \ldots, x_n, c_i^G\}$$

The set Δ_G, and therefore also $\tilde{\Delta_G}$, is totally ordered, left-bounded, and most importantly, discrete. Thus $\tilde{\Delta_G}$ has a minimum we note t_G, and which represents the time that can elapse without G imposing a time blocking. If $\tilde{\Delta_G}$ is empty (i.e., there is to path), then we suppose $t_G = \infty$.

For those synchronizers that are **urgent**, we can now determine the value t_1 as the minimum among the according t_G (and $t_1 = \infty$ if no **urgent** synchronizers exist); thus t_1 represents the time that can elapse before urgency blocks further time elapsing.

In a similar fashion, we can determine the value t_2, which represents the time that can elapse before a **must** time window blocks further time elapsing (which also may be infinite).

We calculate $t_3 \stackrel{\text{def}}{=} min(t_1, t_2)$.

Case $t_3 = 0$: Then the corresponding synchronization is enabled, thus S is not a timelock state.

Case $t_3 > 0$: Then S is relaxed, and a timed transition labelled t' ($t' \leq t_3 \wedge t' < \infty$) can be taken, thus S is not a timelock state.

(iii) Suppose S to be a state in the TLTS of a dense time ATLANTIF module M such that all timed expressions are constants. Following Definition 3.6, we suppose moreover that all transitions that may follow S are only timed transitions. If no such S exists, it is already excluded by the definition that M's TLTS could be strict zeno. Let T_S be the set containing all the timed labels of these outgoing transitions. Because of time additivity (cf. Proposition 4.1), it is sufficient to show that the set T_S is either unbounded or that it has a maximum (and not only a supremum).

But this is already obvious, because as in *(ii)*, we can calculate the time t_3 that can elapse maximally from S on, thus t_3 is this maximum. Therefore, the TLTS of M is not weak zeno.

□

Example 4.9. *In the general case, when timed expressions may contain variables, timelock-freeness is not satisfied by the semantics, as the module of Fig. 4.8 shows:*

module *Timelock_Example* **is dense time**
sync *G* **is** *U1* **end sync**
init *U1*
unit *U1* **is**
  variables V **:int**
  from *S1*
    V := **any int**; G **must in** $[\frac{1}{V}, \frac{1}{V}]$; **stop**
end unit
end module

Figure 4.8: An ATLANTIF specification with a timelock

The initial state $([U1 \mapsto S1], [U1 \mapsto (0, \mathbf{false})], \varnothing)$ *of this module does not allow a discrete transition, because for each value of* V *that can be chosen, the phase of* $U1$ *has to be* $\frac{1}{V}$*, and thus bigger than zero in order to execute the communication action* G **must in** $[\frac{1}{V}, \frac{1}{V}]$. *The initial state does not allow a timed transition either, because for each* $t \in (\mathbb{D} \setminus \{0\})$*, by definition of a dense time domain, there exists an integer* n *such that* $\frac{1}{n} < t$*, thus:*

$$\neg relaxed(([U1 \mapsto S1], [U1 \mapsto (\tfrac{1}{n}, \mathbf{true})], \varnothing)).$$

Therefore, the premise of rule (time) *cannot be satisfied.*

Remark 4.8. *The restriction in the cases* (ii) *and* (iii) *of Proposition 4.4 can be weakened: Instead of only permitting constant timed expressions, we could also allow any timed expression that has a minimal value, because the proofs of cases* (ii) *and* (iii) *basically depend on whether the value* t_3 *can be calculated, which is still possible with the weakened restriction.*

For instance, in Example 4.9, the expression "$\frac{1}{V}$*" does clearly not allow a minimum. If the time window would be "*$[V^2, V^2]$*" instead, then no timelock could occur, because "*V^2*" has the minimum zero. Note that the timelock of Example 4.9 could be avoided if the calculation of* V *and the communication on* G *were in two different discrete steps: Suppose a new synchronizer* H *(visible, hidden, or urgent) and unit* $U1$ *to be extended with two discrete states "***from** *S1* V **:= any int**; H; **to** *S2" and "***from** *S2* G **must in** $[\frac{1}{V}, \frac{1}{V}]$; **stop***", then the minimum could be calculated statically after the communication by* H*.*

In our opinion, the restriction to timed expressions with minimal values hardly excludes any realistic behaviour, but only artificial specifications as the one in the previous example.

Remark 4.9. *Stopping of all units*

It is possible that no unit of an ATLANTIF *module is active (because each unit that was active has been stopped by a synchronizer), which corresponds to the* TLTS *state* $(\varnothing, \varnothing, \varnothing)$*.*

In this case, the premise of rule (time) *is satisfied for all* t*, leading to the same state (as* $\varnothing + t = \varnothing$*), and therefore, an unbounded elapsing of time is possible.*

4.7.3 Analysis of the rules

A syntax definition usually induces an intuitive meaning, which should ideally be congruent with the formal semantics definition. In this section, we show several points concerning this congruence i.e., we discuss whether the semantics rules behave as they are intended to behave.

Consistency and negative premises

In the definition of rule (*time*) (cf. page 86), the hypothesis contains the predicate *relaxed*, which is **true** if certain transitions are *not* derivable. As explained in Remark 3.2 on page 47, such a so-called *negative premise* is inevitable to assure maximal progress of urgent actions.

Regarding the possibility of consistency problems mentioned in Remark 3.2, in [68, 69], a method called *stratification* is proposed to verify that a set of derivation rules containing negative premises is consistent. Concerning real-time formal semantics, [91] reduces stratification to the verification of the following two conditions:

- Rules used to derive discrete transition do not have negative premises, nor premises on timed transitions.
- Rules used to derive timed transitions have only negative premises on discrete transitions.

The ATLANTIF semantics only contain one rule of each type. Rule (*rdv*) depends on the predicate *enabled*, which itself depends on the predicate *synchronizing*. None of these predicates contain negative premises on transitions, or premises on timed transitions in general.

Rule (*time*) depends on the predicate *relaxed*. If *relaxed* is replaced by its definition (cf. page 85), the following writing of the rule is obtained:

$$\frac{\begin{array}{c} t > 0 \wedge (\forall t' < t, G\ v_1 \ldots v_n, \mu, S') \\ \mathit{enabled}((\pi, \theta + t, \rho), G\ v_1 \ldots v_n, \mu, S') \Rightarrow (\neg\mu \wedge \mathit{tag}(G) \neq \textbf{urgent})((\pi, \theta + t', \rho)) \end{array}}{(\pi, \theta, \rho) \xrightarrow{t} (\pi, \theta + t, \rho)}$$

As the logical implication is defined by "$A \Rightarrow B \stackrel{\text{def}}{=} \neg A \vee B$", rule (*time*) thus contains negative premises on the predicate *enabled*. As stated above, *enabled* does not depend on timed transitions. Thus, the two conditions are satisfied, and therefore, the negative premises do not introduce inconsistencies in the ATLANTIF semantics.

Equivalences

Definition 4.1. *Two* ATLANTIF *actions A_1, A_2 are* semantically equivalent*, written $A_1 \approx A_2$, if for each phase t, blocking condition μ, store ρ, local label l and state σ, we have*

$$(A_1, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} \sigma \text{ iff } (A_2, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} \sigma.$$

Proposition 4.5. The sequential composition is associative i.e., for all actions A_1, A_2, A_3:

$$(A_1\,;(A_2\,;A_3)) \approx ((A_1\,;A_2)\,;A_3)$$

(Note that the parentheses are not a syntax construct, but only used here to visualize the statement.)

Proof. The proof is purely technical and quite similar in both directions. Thus, we only show "$\Rightarrow$".

Let d, ρ, l, σ be such that $((A_1\,;(A_2\,;A_3)), d, \rho) \overset{l}{\Longrightarrow} \sigma$.

It has to be shown that $(((A_1\,;A_2)\,;A_3), d, \rho) \overset{l}{\Longrightarrow} \sigma$. Two main cases have to be distinguished:

Case 1: Rule (seq_1) applies to the first composition i.e., $(\exists l_1, l_2, d', \rho')\ (A_1, d, \rho) \overset{l_1}{\Longrightarrow} (\delta, d', \rho')$ (hyp. 1) and $((A_2\,;A_3), d', \rho') \overset{l_2}{\Longrightarrow} \sigma$ (hyp. 2) and $l = l_1 + l_2$. Two subcases can apply for (hyp. 2):

Case 1.1: Rule (seq_1) applies i.e., $(\exists l_3, l_4, d'', \rho'')\ (A_2, d', \rho') \overset{l_3}{\Longrightarrow} (\delta, d'', \rho'')$ (hyp. 3) and $(A_3, d'', \rho'') \overset{l_4}{\Longrightarrow} \sigma$ (hyp. 4) and $l_2 = l_3 + l_4$.
By (hyp. 1) and (hyp. 3), with rule (seq_1) we derive $((A_1\,;A_2), d, \rho) \overset{l_1+l_3}{\Longrightarrow} (\delta, d'', \rho'')$. From this derivation and (hyp. 4), with rule (seq_1) we then derive $(((A_1\,;A_2)\,;A_3), d, \rho) \overset{(l_1+l_3)+l_4}{\Longrightarrow} \sigma$. The operator "+" is associative, thus $(l_1 + l_3) + l_4 = l_1 + (l_3 + l_4) = l$.

Case 1.2: Rule (seq_2) applies i.e., $(\exists s \neq \delta, d'', \rho'')\ (A_2, d', \rho') \overset{l_2}{\Longrightarrow} \sigma$ (hyp. 5) and $\sigma = (s, d'', \rho'')$.
By (hyp. 1) and (hyp. 5), with rule (seq_1) we derive $((A_1\,;A_2), d, \rho) \overset{l_1+l_2}{\Longrightarrow} \sigma$. From this derivation, with rule (seq_2) we derive $(((A_1\,;A_2)\,;A_3), d, \rho) \overset{l_1+l_2}{\Longrightarrow} \sigma$.

Case 2: Rule (seq_2) applies to the first composition i.e., $(\exists s \neq \delta, d', \rho')\ (A_1, d, \rho) \overset{l}{\Longrightarrow} \sigma$ (hyp. 6) and $\sigma = (s, d', \rho')$.
By (hyp. 6), with rule (seq_2) we derive $((A_1\,;A_2), d, \rho) \overset{l}{\Longrightarrow} \sigma$. From this derivation, with further application of rule (seq_2) we derive $(((A_1\,;A_2)\,;A_3), d, \rho) \overset{l}{\Longrightarrow} \sigma$. $\square$

Proposition 4.6. Some time windows can be decomposed.

(i) For each gate G, offers $\bar{O}$ (short for $O_1 \ldots O_n$), and time windows W_1, W_2:

$$G\ \bar{O}\ \textbf{may in}\ W_1\ \textbf{or}\ W_2 \;\approx\; \textbf{select}\ G\ \bar{O}\ \textbf{may in}\ W_1\ [\,]\ G\ \bar{O}\ \textbf{may in}\ W_2\ \textbf{end}.$$

(ii) For each gate G, offers $\bar{O}$, and time windows W_1, W_2, if for all ρ, D_1, D_2 with $win\_eval(W_1, \rho, D_1)$, $win\_eval(W_2, \rho, D_2)$, the inequation $sup(D_1) > sup(D_2)$ holds (where sup is as usual the function mapping to the least upper bound of a set), then:

$$G\ \bar{O}\ \textbf{must in}\ W_1\ \textbf{or}\ W_2 \;\approx\; \textbf{select}\ G\ \bar{O}\ \textbf{must in}\ W_1\ [\,]\ G\ \bar{O}\ \textbf{may in}\ W_2\ \textbf{end}.$$

Proof. Both statements are proved in a similar way. Therefore, we only show the proof of the second statement:

"$\Rightarrow$": Let be $t, \mu, \rho, l, \mu', \rho'$ such that $(G\ \bar{O}\ \textbf{must in}\ W_1\ \textbf{or}\ W_2, (t, \mu), \rho) \overset{l}{\Longrightarrow} (\delta, (t, \mu'), \rho')$.

Then, by hypothesis of rule ($comm$), there exists a set $D \subseteq \mathbb{D}$ such that we have the predicate $win\_eval(W_1\ \textbf{or}\ W_2, \rho, D)$ and $t \in D$.

By definition of the predicate *win_eval*, we therefore know that there exist sets D_1, D_2 that satisfy $\mathit{win\_eval}(W_1, \rho', D_1)$, and $\mathit{win\_eval}(W_2, \rho', D_2)$, and $D = D_1 \cup D_2$. By the condition of the statement, we thus have $\mathit{sup}(D) = \mathit{sup}(D_1)$.

Thus, it is possible that $\mathit{up\_lim}(\textbf{must}, D_1, t)$ is **true**, but $\mathit{up\_lim}(\textbf{must}, D_2, t)$ is necessarily **false**. We can deduce that $\mathit{up\_lim}(\textbf{must}, D, t) = \mathit{up\_lim}(\textbf{must}, D_1, t)$ and that $\mathit{up\_lim}(\textbf{must}, D_2, t) = \mathit{up\_lim}(\textbf{may}, D_2, t)$. This last equation is the key element of the proof.

Clearly, $t \in D_1$ or $t \in D_2$. Thus, by rule (*comm*), we have

$$(G\ \bar{O}\ \textbf{must in}\ W_1, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (\delta, (t, \mathit{up\_lim}(\textbf{must}, D_1, t)), \rho')$$

or we have

$$(G\ \bar{O}\ \textbf{must in}\ W_2, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (\delta, (t, \mathit{up\_lim}(\textbf{must}, D_2, t)), \rho').$$

The second case is equivalent to

$$(G\ \bar{O}\ \textbf{may in}\ W_2, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (\delta, (t, \mathit{up\_lim}(\textbf{may}, D_2, t)), \rho').$$

By definition of rule (*select*), we thus can deduce that

$$(\textbf{select}\ G\ \bar{O}\ \textbf{must in}\ W_1\ \texttt{[]}\ G\ \bar{O}\ \textbf{may in}\ W_2\ \textbf{end}, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (\delta, (t, \mathit{up\_lim}(\textbf{must}, D, t)), \rho').$$

"$\Leftarrow$": Let be $t, \mu, \rho, l, \mu', \rho'$ such that

$$(\textbf{select}\ G\ \bar{O}\ \textbf{must in}\ W_1\ \texttt{[]}\ G\ \bar{O}\ \textbf{may in}\ W_2\ \textbf{end}, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (\delta, (t, \mu'), \rho').$$

Again, by definition of *win_eval*, we know that there exist D_1, D_2 that satisfy $D = D_1 \cup D_2$, $\mathit{win\_eval}(W_1, \rho', D_1)$, and $\mathit{win\_eval}(W_2, \rho', D_2)$.

Also again, we state that $\mathit{sup}(D) = \mathit{sup}(D_1)$ and it is possible that $\mathit{up\_lim}(\textbf{must}, D_1, t)$ is **true**, but $\mathit{up\_lim}(\textbf{must}, D_2, t)$ is necessarily **false**; and thus we can now deduce that $\mathit{up\_lim}(\textbf{must}, D_2, t) = \mathit{up\_lim}(\textbf{may}, D_2, t)$.

By this, we are able to inversely apply the same argumentation as above. □

Proposition 4.7. The action semantics of ATLANTIF satisfy the following:

(i) **wait** $0 \approx$ **null**

(ii) (Associativity of **select**)
Nested **select**-actions behave the same as flattened **select**-actions i.e. for all actions A_1, A_2, A_3:

$$\textbf{select select}\ A_1\ \texttt{[]}\ A_2\ \textbf{end}\ \texttt{[]}\ A_3\ \textbf{end} \approx \textbf{select}\ A_1\ \texttt{[]}\ \textbf{select}\ A_2\ \texttt{[]}\ A_3\ \textbf{end end} \approx \textbf{select}\ A_1\ \texttt{[]}\ A_2\ \texttt{[]}\ A_3\ \textbf{end}.$$

(iii) Actions behind a jump action are "dead code" i.e., for each action A, for each discrete state s:

$$\textbf{to}\ s\texttt{;}\ A \approx \textbf{to}\ s$$

(iv) Actions behind a stop action are also "dead code" i.e., for each action A:

$$\textbf{stop};\ A \approx \textbf{stop}$$

Proof. All statements are consequences of the action semantics. Thus, for each case let t, t' be phases, μ, μ' blocking conditions, ρ, ρ' stores and l a local label.

(i) Clearly, $eval(0, \rho, 0)$ and $t \geq 0 \geq 0$ are always true. Therefore:

$$\begin{array}{ll} & (\textbf{wait}\ 0, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \\ \text{iff} & l = \varepsilon, s = \delta, t = t', \mu = \mu', \rho = \rho' \text{ and } eval(0, \rho, 0) \text{ and } t \geq 0 \geq 0 \\ \text{iff} & l = \varepsilon, s = \delta, t = t', \mu = \mu', \rho = \rho' \\ \text{iff} & (\textbf{null}, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \end{array}$$

(ii) Obviously, the **select** action satisfies commutativity. Therefore, we can restrict ourselves e.g., to the second "$\approx$".

$$\begin{array}{ll} & (\textbf{select}\ A_1\ \texttt{[]}\ \textbf{select}\ A_2\ \texttt{[]}\ A_3\ \textbf{end}\ \textbf{end}, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \\ \text{iff} & (A_1, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \\ & \qquad \text{or } (\textbf{select}\ A_2\ \texttt{[]}\ A_3\ \textbf{end}, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \\ \text{iff} & (A_1, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \text{ or } (A_2, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \\ & \qquad \text{or } (A_3, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \\ \text{iff} & (\textbf{select}\ A_1\ \texttt{[]}\ A_2\ \texttt{[]}\ A_3\ \textbf{end}, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \end{array}$$

(iii) By static semantics, we know that $s \neq \delta$.

$$\begin{array}{ll} & (\textbf{to}\ s;\ A, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \\ \text{iff} & l = \varepsilon, t = t', \mu = \mu', \rho = \rho' \\ \text{iff} & (\textbf{to}\ s, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (s, (t', \mu'), \rho') \end{array}$$

(iv) The same reasoning as in *(iii)* applies here:

$$\begin{array}{ll} & (\textbf{stop};\ A, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (\Omega, (t', \mu'), \rho') \\ \text{iff} & l = \varepsilon, t = t', \mu = \mu', \rho = \rho' \\ \text{iff} & (\textbf{stop}, (t, \mu), \rho) \stackrel{l}{\Longrightarrow} (\Omega, (t', \mu'), \rho') \end{array}$$

□

Remark 4.10. *The seven equivalences stated in the Propositions 4.5, 4.6, and 4.7 have different applications: The equivalence of Proposition 4.5 is necessary to assure that the semantics are well-defined, the equivalences of 4.6 will be refered to in Sections 6.1.4 and 6.2.4, and the equivalences of 4.7 can be seen as indications that the semantics work as intuitively intended.*

Derived Constructs

The ATLANTIF syntax of Table 4.3 is not minimal i.e., there are a few constructs that could be deleted from the syntax definition without loss of expressive power, because it is possible to derive them from other constructs. The most important cases are shown in the following propositions and remarks.

Proposition 4.8. The **if** action can be derived from the **case** action. Formally, for A_1, A_2 actions and E an expression of boolean type:

$$\textbf{if } E \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ end} \stackrel{\text{def}}{=} \textbf{case } E \textbf{ is true} \texttt{ -> } A_1 \texttt{ | } \textbf{any bool} \texttt{ -> } A_2 \textbf{ end}$$

Proof. It has to be shown that left-hand-side and right-hand-side of the above definition are semantically equivalent. Let therefore d be a phase and blocking condition, ρ be a store, l a local label, and σ a local state. Two cases have to be distinguished:

(i) case $eval(E, \rho, \textbf{true})$: Then we can derive the following:

$$\begin{array}{ll} & (\textbf{if } E \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ end}, d, \rho) \xrightarrow{l} \sigma \\ \text{iff} & (A_1, d, \rho) \xrightarrow{l} \sigma \\ \text{iff} & match(\textbf{true}, \rho, \textbf{true}) = \rho \neq \textbf{fail} \text{ and } (A_1, d, \rho) \xrightarrow{l} \sigma \\ \text{iff} & (\textbf{case } E \textbf{ is true} \texttt{ -> } A_1 \texttt{ | } \textbf{any bool} \texttt{ -> } A_2 \textbf{ end}, d, \rho) \xrightarrow{l} \sigma \end{array}$$

(ii) case $eval(E, \rho, \textbf{false})$: Then we can derive the following:

$$\begin{array}{ll} & (\textbf{if } E \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ end}, d, \rho) \xrightarrow{l} \sigma \\ \text{iff} & (A_2, d, \rho) \xrightarrow{l} \sigma \\ \text{iff} & match(\textbf{false}, \rho, \textbf{true}) = \textbf{fail} \text{ and } match(\textbf{false}, \rho, \textbf{any bool}) = \rho \neq \textbf{fail} \\ & \qquad\qquad \text{and } (A_2, d, \rho) \xrightarrow{l} \sigma \\ \text{iff} & (\textbf{case } E \textbf{ is true} \texttt{ -> } A_1 \texttt{ | } \textbf{any bool} \texttt{ -> } A_2 \textbf{ end}, d, \rho) \xrightarrow{l} \sigma \end{array}$$

□

Remark 4.11. *The* **stop** *action is a derived construct: Supposing a unit u in which "terminus" is not among the identifiers of discrete states i.e., terminus* $\notin \mathcal{S}_u$*. Then, if a new discrete state*

from *terminus* **null**

*is added, then in the context of this unit, a jump "***to** *terminus" behaves like a "***stop***". We do not have* **stop** $\approx$ **to** *terminus (because they lead to different states), but given that* $act(\Omega) = act(terminus) = \textbf{null}$*, this difference is irrelevant.*

In particular, for $d = (0, \textbf{false})$ *and any* ρ*, in both cases we can only derive a single transition on the unit level (by application of rule (*null*)):*

$$(\textbf{null}, d, \rho) \stackrel{\varepsilon}{\Longrightarrow} (\delta, d, \rho)$$

Thus, if the unit u is in a discrete state like Ω*, it cannot occur in the synchronization set of a synchronization as described by the predicate* synchronizing*. By definition of the* *predicate* relaxed*, only the possibility of synchronizations may block the elapsing of time, therefore u may idle indefinitely.*

Clearly, it is always possible to define an additional discrete state like terminus, therefore the statement that **stop** *is a derived construct holds.*

Remark 4.12. *The* **among** *construct in synchronization formulas (cf. Section 4.5.3) is a derived construct. This is obvious from the formal definition of the set sync(G) for a synchronizer G, where the dynamic semantics of a synchronization formula using* **among** *is given based on synchronization formulas built from* **and** *and* **or** *formulas. Thus, an* **among***-formula is purely for conveniently expressing complex synchronizations.*

Remark 4.13. *Variable initialization is not a derived construct.*

Suppose for instance the following unit:

> **unit** *Sender* **is**
> **variables** *Package*:**int** := 3
> **from** *Ready*
>   *Transmission* !*Package* **in** [2, . . . [; **stop**
> **end unit**

In this case, it is possible to construct unit Sender *in another way, without the variable initializations:*

> **unit** *Alternative_Sender* **is**
> **variables** *Package*:**int**
> **from** *before_Ready*
>   *Package* := 3; **to** *Ready*
> **from** *Ready*
>   *Transmission* !*Package* **in** [2, . . . [; **stop**
> **end unit**

*It is clear from the semantics definition that from both units we can derive the same semantics: a succession of timed transitions of duration at least two, possibly followed by a discrete transition labelled "*Transmission *3".*

However, in the general case it is not possible to replace variable initializations by such a preliminary discrete state, as it can be seen from the following extension of the initial unit with two subunits:

```
unit Extended_Sender is
variables Package:int := 3
from Ready
    Transmission !Package in [2,...[; stop

    unit Sub_Sender_1 is
    from Ready1
        Check !Package; stop
    end unit
    unit Sub_Sender_2 is
    from Ready2
        Compare !Package; stop
    end unit

end unit
```

If we suppose now the module to be defined with Extended_Sender *as an initial unit, with another initial unit* Controller*, and with the following synchronizer:*

sync *Activate_Sub_Senders* **is** *Controller* **stop** *Extended_Sender*
start *Sub_Sender_1* **,** *Sub_Sender_2* **end sync**

Then a preliminary discrete state as in Alternative_Sender *would not be allowed by the static semantics, as such a module would not be well-initialized (cf. Section 4.6.2): If the synchronization* Activate_Sub_Senders *occurs before* Transmission*, then the variable* Package*, would not be initialized although it would need to be.*

Neither is it possible to define a preliminary state in each of the units Extended_Sender*,* Sub_Sender_1*, and* Sub_Sender_2*, because then the variable* Package *would not be well-accessible (cf. Section 4.4.4 and the algorithm A.3 on page 220): Both* Sub_Sender_1 *and* Sub_Sender_2 *can be active at the same time, thus it is not permitted that they both define* Package*.*

Thus, because of the possibility accessing variables in subunits and the related restrictions in the static semantics, it is not possible to derive the variable initialization.

Further derivable constructs. There are other syntax constructs that do not occur in the definition of Table 4.3, but that could easily be derived from the existing ones. We show two examples:

- **for** loops can be derived, for instance in the following way:

$$\textbf{for } V \textbf{ in } E_1 \,..\, E_2 \textbf{ do } A_0 \textbf{ end } [\textbf{for}] \stackrel{\text{def}}{=} V := E_1; \quad \textbf{while } V \leq E_2 \textbf{ do } A_0;\ V := V+1 \textbf{ end}$$

  We suppose V, E_1, E_2 to be of integer type.

- It is also possible to derive **if** actions with multiple branches, for instance in the following way:

$$\begin{array}{l}\textbf{if } E_1 \textbf{ then } A_1 \textbf{ elseif } E_2 \textbf{ then } A_2 \ \ldots \ \textbf{elseif } E_n \textbf{ then } A_n \textbf{ else } A_{n+1} \textbf{ end} \stackrel{\text{def}}{=} \\ \qquad \textbf{if } E_1 \textbf{ then } A_1 \textbf{ else if } E_2 \textbf{ then } A_2 \ \ldots \\ \qquad \textbf{else if } E_n \textbf{ then } A_n \textbf{ else } A_{n+1} \underbrace{\textbf{end} \ \ldots \ \textbf{end}}_{n \text{ times}}\end{array}$$

  We suppose all expressions to be of boolean type and $n \geq 2$.

Remark 4.14. *Undecidable conditions*

Supposing a more detailed definition of the data part of Atlantif*, a more sophisticated definition of the static semantics could be established, where only decidable or only efficiently decidable condition expressions are allowed. For instance, there is presumably no algorithm to decide whether the action*

$$V_1, V_2, V_3, n := \textbf{any int}, \textbf{int}, \textbf{int}, \textbf{int where } V_1 > 0 \wedge V_2 > 0 \wedge V_3 > 0 \wedge n > 2 \wedge ({V_1}^n + {V_2}^n = {V_3}^n)$$

can be executed[15]*. A possible solution would be a restriction of condition expressions to Presburger arithmetics [101].*

Remark 4.15. *In Section 4.4.4, we stated that variable identifiers have to be globally unique. This restriction is due to two reasons: First, without this restrictions there could be overlapping and thus ambiguity in the scopes of two variables. Second, stores are defined on variable identifiers; the store of a global state cannot have two equally named variables in its domain.*

The second reason can be overcome, permitting us to weaken the restriction. To this end, stores have to be defined in another way that assures well-definedness: Such extended stores *are partial functions $\tilde{\rho} : \mathcal{V} \times \mathbb{U} \rightarrow Val$, where a tuple (V, u) can only be in the domain of $\tilde{\rho}$, if $V \in decl(u)$. Then, it is sufficient to require that if two variables with the same identifier V are declared in two different units u, u', then we have $u \not\preceq u'$ and $u' \not\preceq u$.*

Given that the restriction to unique names for variables is however without any impact on the expressive power of Atlantif*, we keep it for the sake of simplicity.*

4.8 Conclusion

4.8.1 Suitability as an intermediate format

High-level syntax constructs and semantics. In Section 3.3, we analysed a wide range of high-level constructs. Our choices for Atlantif are based on this analysis, aimed to cover a reasonably large part of this range. For instance, life reducers, communication-

[15] Observed from a theoretical point of view. In practice, formal verification tools like the Cadp toolbox would simply enumerate a limited number of possible values and then conclude that there is no solution.

independent delays, flexible offers, intuitive but expressive synchronization, etc. were integrated in ATLANTIF.

Nevertheless, some constructs that sometimes appear in high-level languages were deliberately not introduced in ATLANTIF, for example the following:

- time capture variables or clocks (both somehow similar concepts)
- arbitrary combination of sequential and parallel composition
- recursion

The reason for not implementing such ideas is that the language should not be overloaded with scarcely intuitive constructs, and regarding usability, intuition is more important than maximal expressive power. Moreover, the expressive power is not gravely reduced without such constructs as the Chapters 5 and 7 will illustrate.

In addition, ATLANTIF is intended to be semantically close to high-level languages, in particular, we continued the approach of [61] to provide big-step semantics i.e., semantics that only define discrete transitions for synchronizations of communication actions, and not for technical events such as assignments or the evaluation of conditions. This is achieved by three different mechanisms that eliminate ε-transitions: the rule (seq_1) (page 69), the rule (ε-*elim*) (page 73), and the predicate *enabled* (page 85). This is a central condition for the preservation of the semantics we required in the introduction.

Graphical model structures. Like in most graphical models, ATLANTIF has a structure of processes (units) with discrete states, which are defined independently of interactions. This enables us to define translations to graphical models in a quite natural way, as the intuition behind an ATLANTIF discrete state is practically the same as behind the location of a timed automaton or the place in a time Petri net (as we will see in more detail in Chapter 6). Moreover, it is also an intuitive approach that we will use to split up the concise multibranch transitions into "condition/action" transitions as they appear in graphical models.

4.8.2 Possible extensions

Not everything that can be expressed in some high-level or intermediate languages can be expressed in ATLANTIF. This section presents several ideas on how our model could be extended, each time with a discussion of whether we consider this extension desirable.

Constants. Declaring constants in the file containing an ATLANTIF module, to which expressions can refer, would obviously be very convenient in order to modify or to reuse parts of this module. Such an extension would be a stricly technical one, as there is no real impact on the semantics or the expressive power.

Indeed, the tool implementation of Atlantif allows the user to define constants along with the definitions of types and functions. This is already used in the example module of Chapter 7 (cf. Fig 7.1). An exact syntax definition for constants can be found in Appendix C.

Priorities. As explained in Section 3.4, other intermediate models enable the definition of priorities e.g., Fiacre and Bip. For the time being, such a construct does not exist in Atlantif.

Several approaches seem possible to implement priorities e.g.:

- on the module level, with a partial order between the synchronizers (such is the approach of Fiacre and most other models),
- within each synchronizer, by a new operator in the synchronization formula that induces a partial order on the synchronization sets,
- or within the action syntax, by a modified **select** that defines an order on its alternative branches.

It remains to be seen whether priorities are indeed useful, and which variant should be chosen, depending on applications of Atlantif.

Broadcast. Broadcast synchronizations exist e.g., in the Uppaal TA dialect (cf. Section 3.2.2) and in the Bip model (cf. Section 3.4.2). They do not exist in Atlantif, because we decided our synchronization would be undirected instead of implementing a directed synchronization (i.e., where a sender and a receiver can be identified); thus the intuition behind our definition of offers could not be coherent with such an extension.

Note that behaviour *similar* to a broadcast on a channel G from a unit S, receivable from the units $R1$ to Rn could be expressed as follows:

$$\textbf{sync } G \textbf{ is } S \textbf{ and } 0 \textbf{ or } \ldots\ n \textbf{ among } (R1, \ldots, Rn) \textbf{ end sync}$$

This does not behave exactly like the broadcast of Uppaal or Bip, as it does not necessarily synchronize a *maximum* of receivers. This could be achieved using priorities (with the second of the approaches above).

Chapter 5

Translating high-level constructs into ATLANTIF

Abstract In this chapter, we present how several typical high-level constructs from different languages may be represented in Atlantif, either in a general fashion, or with a concrete example.

We also show informally that the semantics of the high-level constructs are preserved by the Atlantif representation.

5.1 Introduction

In this chapter, we will analyze the expressive power of Atlantif by giving several examples of constructs typically provided by high-level languages, followed in each case by one possible solution for an Atlantif module with the same behaviour. The choice of constructs discussed covers a broad range of aspects related to concurrency and real time (data-related aspects and corresponding examples have already been discussed in [61]).

It is not our objective here to give formal translations from high-level languages to Atlantif, but instead to informally demonstrate that Atlantif has indeed a high expressive power, which is close to high-level languages. This is a key feature and desirable in any intermediate model, because this adjacency may ultimatively enable us to perform systematic translations of high-level constructs, notably constructs used in Lotos NT and in E-Lotos. Moreover, high expressive power makes Atlantif also appropriate as an alternative specification language.

It should be noted that the exact behaviours of some of the constructs discussed in the remainder of this chapter depend on the context in which they occur. This means firstly that not all of our translations can be generalized in a "mechanical" and simple way. Secondly, it also means that we will not be able to give formal proofs of our translations being correct, because the semantics of Atlantif is defined only for complete modules. Instead, we will reason schematically and on a more informal level to show the stability of the semantics during the translation.

In the remainder of this chapter, we will sometimes mention possible derivations of transitions *modulo equivalence by time additivity* (or shorter "modulo time additivity"). Given two finite paths whose last transitions are discrete and all other transitions are timed, we say that they are equivalent by time additivity if the following are equal in both paths: their first states, their penultimate states, their final states, the sums of their timed labels, and their discrete labels. Formally, two paths

$$S_1 \xrightarrow{t_1} \ldots \xrightarrow{t_n} S_2 \xrightarrow{a} S_3 \text{ and } S_1' \xrightarrow{t_1'} \ldots \xrightarrow{t_{n'}'} S_2' \xrightarrow{a'} S_3'$$

are equivalent if $S_1 = S_1'$, $S_2 = S_2'$, $S_3 = S_3'$, $\Sigma_{i \in 1..n} t_i = \Sigma_{i \in 1..n'} t_i'$, and $a = a'$. Clearly, if one such finite path is derivable, by Proposition 4.1 (page 90) each path equivalent by time additivity is also derivable. Thus, we make an abstraction that enables us to discuss possibly infinite sets of paths by a single representative.

5.2 Combination of sequential and parallel composition

5.2.1 Statement

Some high-level languages enable sequential and parallel composition to be freely combined, for instance the process shown in Fig. 5.1, written in E-LOTOS.

```
process Semi_Ordered_Redistribution [In:package, Out_1a:package,
                                     Out_1b:package, Out_2:package] is
var X:package in
  In (? X:package);
  (Out_1a (!X)
    |||
  Out_1b (!X) );
  Out_2 (!X)
end var
end process
```

Figure 5.1: E-LOTOS code for semi-ordered redistribution

The process *Semi_Ordered_Redistribution*, inspired from a similar example in [122], describes a protocol for data transmission. First, the process receives a value of the data type "*package*" on the gate "*In*", then makes a redistribution in three directions: It begins with one of the two gates "*Out_1a*" or "*Out_1b*", followed by the other one, and finishes with an output on the gate "*Out_2*".

5.2.2 Translation to ATLANTIF

ATLANTIF, in contrast, does not enable parallel composition within a unit (cf. Section 4.4.1). We translate the process *Semi_Ordered_Redistribution* into ATLANTIF by creating one unit for each fragment of sequential behaviour and by defining synchronizers that start and/or stop these units, which controls the parallel composition. The resulting code is shown in Fig. 5.2.

In this translation, we have defined one unit around each of the four communications. First, only *Main_Unit* is active. When it synchronizes with the environment on *In* and a value in X is received, control is passed to the units *Direction_1* and *Direction_2*. After each one of these two units has synchronized on its gate (*Out_1a* and *Out_1b* respectively), they pass control to the unit *Direction_3* by a silent synchronization on *CP*. With a synchronization on *Out_2*, the behaviour ends.

This example has a longer description in ATLANTIF than in E-LOTOS. This is due to the composition of sequential and parallel behaviour, which is necessarily difficult to implement in an automata-based model i.e., a model that represents sequential behaviour on one level and then parallel behaviour on a higher level.

Correctness of the translation. Following E-LOTOS semantics [83], the discrete transitions that can be derived from the process *Semi_Ordered_Redistribution* are the following: a transition labelled "*In v*" (for v an arbitrary value from the sort of *package*), followed by two transitions labelled "*Out_1a v*" and "*Out_1b v*" (in any order), and finally a transition labelled "*Out_2 v*". As the scope of X ends when this action ends, the state reached by that last discrete transition has no value for X. We now explain that the ATLANTIF semantics define the same behaviour, by giving details on two essential aspects for value assignments on X. Note before that by the semantics definition, we have

$$\mathit{accessible}(X) = \{\mathit{Main\_Unit}, \mathit{Direction\_1}, \mathit{Direction\_2}, \mathit{Direction\_3}\}.$$

Thus, whenever at least one of these four units is active, X can have a value assigned.

First we show that the variable X keeps its value during the control passage by the synchronizer *In*. By the unit semantics we can derive in *Main_Unit* the following transition:

$$(\mathit{act}(s1a), (t_0, \mathbf{f}), \varnothing) \xRightarrow{\mathit{In}\ v} (\Omega, (t_0, \mathbf{f}), [X \mapsto v])$$

t_0 is an arbitrary time value and v is an arbitrary value generated by the reception offer; the mapping $[X \mapsto v]$ is added to the store by application of the function *accept*.

Then we can apply the predicate *next_ρ* as follows:

$$\mathit{next\_}\rho(\rho, [X \mapsto v], \{\mathit{Direction\_1}, \mathit{Direction\_2}\}, \mathit{In}, \rho')$$

which by definition is equivalent to

$$\begin{aligned} \rho' = \;& ((\rho \oslash [X \mapsto v]) \ominus \{V \mid (\forall u \in \mathcal{U})\ u \notin \mathit{accessible}(V)\}) \\ & \oslash [V \mapsto \rho_0(V) \mid V \in \mathit{dom}(\rho_0) \wedge (\exists u \in \mathit{start}(\mathit{In}))\ V \in \mathit{decl}(u)] \\ = \;& ((\varnothing \oslash [X \mapsto v]) \ominus \varnothing) \oslash \varnothing \\ = \;& [X \mapsto v]. \end{aligned}$$

With this, we can apply the predicate *synchronizing* to represent the first synchronization as follows:

$$\mathit{synchronizing}(((\pi, \theta, \rho), \varnothing), \mathit{In}\ v, \mathbf{f}, ((\pi', \theta', \rho'), \varnothing))$$

We have $\pi = [\mathit{Main\_Unit} \mapsto s1a]$, $\pi' = [\mathit{Direction\_1} \mapsto s112b, \mathit{Direction\_2} \mapsto s113b]$,

```
module Semi_Ordered_Redistribution is
dense time
type package is ...
sync In is Main_Unit stop Main_Unit start Direction_1 , Direction_2 end sync
sync Out_1a is Direction_1 end sync
sync Out_1b is Direction_2 end sync
sync Out_2 is Direction_3 end sync
sync CP :silent is Direction_1 and Direction_2
            stop Direction_1 , Direction_2 start Direction_3 end sync

init Main_Unit

unit Main_Unit is
variables X :package
  from s1a
    In ?X ; stop

  unit Direction_1 is
    from s112a
      Out_1a !X ; to s112b
    from s112b
      CP ; stop
  end unit -- Direction_1
  unit Direction_2 is
    from s113a
      Out_1b !X ; to s113b
    from s113b
      CP ; stop
  end unit -- Direction_2
  unit Direction_3 is
    from s11a
      Out_2 !X ; reset X ; stop
  end unit -- Direction_3

end unit -- Main_Unit
end module
```

Figure 5.2: ATLANTIF translation for semi-ordered redistribution

$\rho = \varnothing$ and $\rho' = [X \mapsto v]$ as derived before; θ, θ' are not relevant here. Therefore:

$$(\pi, \theta, \varnothing) \xrightarrow{In\ v} (\pi', \theta', [X \mapsto v])$$

Thus X is indeed defined after the control passage.

Second we show that after the synchronization on *Out_2*, the variable X has no longer a value assigned. As above, in the unit *Direction_3* the following transition can be derived (showing this time how the **reset** deletes the assignment to X):

$$(act(s11a), (t_1, \mathbf{f}), [X \mapsto v]) \overset{Out\_2\ v}{\Longrightarrow} (\Omega, (t_1, \mathbf{f}), \varnothing)$$

This finally leads to the TLTS transition:

$$\begin{array}{r}([Direction\_1 \mapsto s112b, Direction\_2 \mapsto s113b], \theta'', [X \mapsto v]) \xrightarrow{Out\_2\ v} \\ ([Direction\_3 \mapsto \Omega], [Direction\_3 \mapsto (0, \mathbf{f})], \varnothing)\end{array}$$

Note that θ'' depends on how much time elapsed, which we can ignore as there are no time constraints. We can see that in the state following the transition, X has no value assigned.

5.3 Delay and timed communication

5.3.1 Statement

In this section, we compare the real-time constructs of Atlantif with the real-time constructs of E-Lotos. We consider the E-Lotos process of Fig. 5.3, which describes the nondeterministic choice between two gates G and H. Gate G is hidden and delayed by eight time units. Note that in E-Lotos, communication on hidden gates automatically becomes urgent. A communication on gate H, due to its time constraint, can occur only if more than five time units have elapsed. Note that the time capture construct of E-Lotos used in this example is very similar to ET-Lotos (cf. Section 3.3.3).

5.3.2 Translation to ATLANTIF

The translation to Atlantif of the example of Fig. 5.3 is given in Fig. 5.4. It uses real-time constructs of Atlantif very similar to E-Lotos.

In this translation, the urgency of G is represented by an **urgent** synchronizer. Note that there is little difference in size between the E-Lotos and the Atlantif code.

Correctness of the translation. Modulo time additivity, two sucessions of transitions can be derived from the Atlantif code. First, for $t_1 \geq 8$, we can derive in *U1* that $(act(S1), (t_1, \mathbf{f}), \varnothing) \overset{G}{\Longrightarrow} (\Omega, (t_1, \mathbf{f}), \varnothing)$, which leads to the following:

```
process Delay_and_Timed_Communication [G:none, H:none] is
(
  hide G:none in
    wait 8; G
  end hide
[]
  var X:time in H @?X [X > 5] end var
)
end process
```

Figure 5.3: E-Lotos code for different real-time constructs

```
module Delay_and_Timed_Communication is
dense time
sync G : urgent is U1 end sync
sync H is U1 end sync
unit U1 is
  from S1
    select
            wait 8; G; stop
        []
            H in ]5, . . . [; stop
    end select
end unit
end module
```

Figure 5.4: Atlantif translation for Fig. 5.3

$$([U1 \mapsto S1], [U1 \mapsto (0, \mathbf{f})], \varnothing) \xrightarrow{t_1} ([U1 \mapsto S1], [U1 \mapsto (t_1, \mathbf{f})], \varnothing) \xrightarrow{\tau} ([U1 \mapsto \Omega], [U1 \mapsto (0, \mathbf{f})], \varnothing)$$

Second, for $t_2 > 5$, we can derive in $U1$ that $(act(S1), (t_2, \mathbf{f}), \varnothing) \stackrel{H}{\Longrightarrow} (\Omega, (t_2, \mathbf{f}), \varnothing)$, which leads to the following:

$$([U1 \mapsto S1], [U1 \mapsto (0, \mathbf{f})], \varnothing) \xrightarrow{t_2} ([U1 \mapsto S1], [U1 \mapsto (t_2, \mathbf{f})], \varnothing) \xrightarrow{H} ([U1 \mapsto \Omega], [U1 \mapsto (0, \mathbf{f})], \varnothing)$$

Clearly, these are exacly the same two successions of transitions derivable in the E-LOTOS process.

5.4 Latency

5.4.1 Statement

The expressive power of ATLANTIF is not restricted to constructs occurring in E-LOTOS or in its dialect LOTOS NT. To illustrate this, this section discusses *latency*, a concept elaborated with the language RT-LOTOS [48]. In the general case, latency behaviour cannot be expressed in E-LOTOS or LOTOS NT.

The following fragment of RT-LOTOS code shows a typical case of latency:

```
...
hide G in
  (delay(3) latency(2) G;
  stop)
...
```

Semantically, this example begins with the elapsing of three time units ("**delay**(3)"). Then, the communication on G can occur at any time during the following 2 time units ("**latency**(2) G"). As G is hidden, this communication becomes a τ-action. Note that the urgency that comes with the hiding does not influence the latency time; instead G becomes not urgent until the 2 time units have elapsed.

5.4.2 Translation to ATLANTIF

It is not practicable to declare G as an **urgent** synchronizer (as we did for the **hidden** G in Section 5.3). Thus, we have to declare G as a **hidden** synchronizer. To ensure the communication at the last instant possible after a period where it is optional, we use the **must** modality in the communication action, which produces the following ATLANTIF code fragment:

```
...
sync G:hidden is U1 end sync
...
unit U1 is
   from S1
      wait 3;
      G must in [0,2];
      stop
end unit
...
```

It can be seen that this translation has a very similar syntax structure as the initial RT-LOTOS code.

Correctness of the translation. Modulo time additivity, again two successions of two transitions each can be derived from the ATLANTIF code. First, for $3 \leq t_1 < 5$, we can derive that $(act(S1), (t_1, \mathbf{f}), \varnothing) \overset{G}{\Longrightarrow} (\Omega, (t_1, \mathbf{f}), \varnothing)$, which leads to the following:

$$([U1 \mapsto S1], [U1 \mapsto (0, \mathbf{f})], \varnothing) \xrightarrow{t_1} ([U1 \mapsto S1], [U1 \mapsto (t_1, \mathbf{f})], \varnothing) \xrightarrow{\tau} ([U1 \mapsto \Omega], [U1 \mapsto (0, \mathbf{f})], \varnothing)$$

Second, for $t_2 = 5$, we can derive that $(act(S1), (5, \mathbf{f}), \varnothing) \overset{G}{\Longrightarrow} (\Omega, (5, \mathbf{t}), \varnothing)$, which leads to the following:

$$([U1 \mapsto S1], [U1 \mapsto (0, \mathbf{f})], \varnothing) \xrightarrow{5} ([U1 \mapsto S1], [U1 \mapsto (5, \mathbf{t})], \varnothing) \xrightarrow{\tau} ([U1 \mapsto \Omega], [U1 \mapsto (0, \mathbf{f})], \varnothing)$$

Note that this example illustrates once more the mode of operation of the blocking condition, which becomes true after the elapsing of five time units. Note further that for this reason it is not possible that more than five time units elapse.

5.5 Synchronization vectors and generalized parallel composition

5.5.1 Statement

This section shows how synchronization vectors and the generalized parallel composition (GPC) operator defined in [64] can be expressed in ATLANTIF. Unlike for the other constructs of this chapter, we give here general translations rather than illustrations by examples.

gate par GE_0^0 `*` ... `*` GE_0^n `->` G'_0, ..., GE_m^0 `*` ... `*` GE_m^n `->` G'_m **in** B_0 `||` ... `||` B_n **end par**	$GE ::= G \mid$ _

Figure 5.5: Syntax of synchronization vectors (EXP.OPEN 2.0)

Synchronization vectors. Synchronization vectors combine a concise notation of synchronization possibilites with renaming of gates, provided for instance in the EXP.OPEN 2.0 language [86], using the syntax given in Fig. 5.5.

In this definition, each G and each G'_i denotes a gate, and each B_j a process. Semantically, this operator means that the processes B_0 to B_n are composed in parallel and that each of the lines "GE_i^0 `*` ... `*` GE_i^n `->` G'_i" ($i \in 0..m$) describes a possible synchronization. For each $j \in 0..n$, if GE_i^j has the form "_", then the process B_j does not participate in this synchronization. Otherwise (i.e., GE_i^j is a gate G), then the process B_j can participate in this synchronization each time it is ready to perform a communication on G. When all the B_j for which the latter case applies are ready, the synchronization can be performed, resulting in a discrete transition labelled G'_i.

Generalized parallel composition. Representing a further generalization of synchronization vectors, the GPC operator of [64] offers an even more concise notation for parallel composition with complex synchronization possibilities (thus, the definitions always have similar objectives to those we had for the synchronizers in Atlantif). It is defined as an extension to Lotos and has been used in the definitions of Lotos NT and E-Lotos (cf. Section 3.3.4).

We present the GPC operator in two steps (only briefly; a detailed description can be found in [64]). In a simplified version, it takes the form shown in Fig. 5.6.

par
  $\widehat{G_1}$ `->` B_1
`||` $\widehat{G_2}$ `->` B_2
`||` ...
`||` $\widehat{G_n}$ `->` B_n
endpar

Figure 5.6: Generalized parallel composition – simplified

Each $\widehat{G_i}$ is a set of gates, each B_i a process. Semantically, the processes B_1 to B_n are composed in parallel and a synchronization of the processes $B_{i_1}, \ldots, B_{i_k}$ ($1 \leq i_1 < \ldots < i_k \leq n$) on a gate G is possible if the following are both true:

- $(\forall i' \in \{i_1, \ldots, i_k\})\ G \in \widehat{G_{i'}}$
- $(\forall i' \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\})\ G \notin \widehat{G_{i'}}$

In its complete version, the GPC operator is defined as in Fig. 5.7.

$$\begin{array}{ll}
\textbf{par}\ H_1\#m_1\,\textbf{,}\ \ldots\ \textbf{,}H_p\#m_p\ \textbf{in} & \\
\quad\ \ \widehat{G_1}\ \texttt{->}\ B_1 & \\
\texttt{||}\ \ \widehat{G_2}\ \texttt{->}\ B_2 & \\
\texttt{||}\ \ \ldots & \\
\texttt{||}\ \ \widehat{G_n}\ \texttt{->}\ B_n & \\
\textbf{endpar} &
\end{array}$$

Figure 5.7: Generalized parallel composition – complete

$H_1, \ldots, H_p$ are gates *not* occurring in any of the $\widehat{G_1}, \ldots, \widehat{G_n}$, and $m_1, \ldots, m_p$ are natural numbers in $1..n$. The complete version strictly extends the simplified version: For each gate H_i, a synchronization on H_i is possible between any combination of m_i processes among $B_1, \ldots, B_n$.

5.5.2 Translation to ATLANTIF

Both operators have a twofold function: First, they put processes in parallel; second, they describe how synchronization is possible between these processes. In Atlantif, units are put in parallel by being active at the same time, thus no further translation of parallel composition is necessary. Instead, we describe here how to represent the synchronization possibilities by Atlantif synchronizers. We will suppose that the processes $B_0, \ldots, B_n$ used in the constructs have been translated to Atlantif units with the same identifiers.

Synchronization vectors. For our translation, we have to respect the following restriction: For each line "GE_i^0 `*` ... `*` GE_i^n `->` G_i'" ($i \in 0..m$), we suppose that for each $j \in 0..n$, the term "GE_i^j" has either the form "_" or it denotes the gate G_i'. This means that we cannot translate gate renaming, which is not supported by Atlantif.

However, in some cases this restriction has no impact, if renaming can be used: For each process B_j, each gate among $GE_0^j, \ldots, GE_n^j$ may occur in different lines. If all these lines define the same gate G_i', the model could also be defined with those $GE_{i'}^j$ renamed to G_i' within B_j. If on the other hand a gate of a process appears in two lines with different gates G_{i_1}', G_{i_2}', a translation by this schema would not be possible, but then a more appropriate approach may exist.

For each gate G among $G_0', \ldots, G_m'$, we create one synchronizer i.e., if there are i, j such that $i \neq j$ and $G_i' = G_j'$, both the ith and the jth line are represented within the same synchronizer. At first, we suppose that there is one line "GE_i^0 `*` ... `*` GE_i^n `->` G_i'" such that $G = G_i'$. Then, the synchronizer takes the form

$$\textbf{sync } G \textbf{ is } B_{j_1} \textbf{ and } \ldots \textbf{ and } B_{j_k} \textbf{ end sync},$$

where $0 \leq j_1 < \ldots < j_k \leq m$ and $\{j_1, \ldots, j_k\} = \{j' \mid GE_i^{j'} = G\}$ (i.e., $j_1, \ldots, j_k$ are exactly the indices of the processes synchronizing on G).

Now we suppose that there are several lines "GE_i^0 `*` $\ldots$ `*` GE_i^n `->` G_i'" such that $G = G_i'$. Then we create the synchronization formulas

$$(B_{j_1} \textbf{ and } \ldots \textbf{ and } B_{j_k}), \ldots, (B_{j_1'} \textbf{ and } \ldots \textbf{ and } B_{j_k'})$$

as above, and the synchronizer takes the following form:

$$\textbf{sync } G \textbf{ is } (B_{j_1} \textbf{ and } \ldots \textbf{ and } B_{j_k}) \textbf{ or } \ldots \textbf{ or } (B_{j_1'} \textbf{ and } \ldots \textbf{ and } B_{j_k'}) \textbf{ end sync}$$

Generalized parallel composition. For simplicity, our translation for the GPC operator of Fig. 5.7 uses three steps.

First, we construct the synchronizers for those gates G that occur in at least one of the $\widehat{G_1}, \ldots, \widehat{G_n}$ (i.e., those gates that can be represented by the simplified version of Fig. 5.6). For each of them we define one synchronizer of the form

$$\textbf{sync } G \textbf{ is } (B_{i_1} \textbf{ and } \ldots \textbf{ and } B_{i_k}) \textbf{ or } 1 \textbf{ among } (B_{j_1}, \ldots, B_{j_l}) \textbf{ end sync},$$

where:

- $\{i_1, \ldots, i_k\} \stackrel{\text{def}}{=} \{i \mid G \in \widehat{G_{i'}}\}$
- $\{j_1, \ldots, j_l\} \stackrel{\text{def}}{=} \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$

Note that those are precisely the conditions for a synchronization given above in the GPC semantics. Note also that the "**or** 1 **among** $(B_{j_1}, \ldots, B_{j_l})$" illustrates a difference between our approach (which also corresponds to the approach of synchronization vectors) and the approach of [64]: We consider that each gate communication has to be defined explicitly, including those of a single process (unit) with its environment, whereas the latter are considered to be always possible in [64]. Thus, our synchronizers generally state which synchronizations are accepted by the environment.

Second, we construct the synchronizers for those gates H that occur among the $H_1, \ldots, H_p$ (this can be done independently of the first step, since these two sets of gates are disjoint). For each of them we define one synchronizer of the form

$$\textbf{sync } H \textbf{ is } m_1' \textbf{ or } \ldots \textbf{ or } m_q' \textbf{ among } (B_1, \ldots, B_n) \textbf{ end sync},$$

where the occurrences of H in the first line of the GPC operator are $H\#m_1', \ldots, H\#m_q'$.

Third, for each H' that occurs as a gate in one of the $B_1, \ldots, B_n$ but did not induce a synchronizer by the first or the second step (i.e., it does not occur in the GPC construct itself) we construct a synchronizer of the form

$$\textbf{sync } H' \textbf{ is } 1 \textbf{ among } (B_1, \ldots, B_n) \textbf{ end sync}.$$

As in the first step, where we added the "**or** 1 **among** $(B_{j_1}, \ldots, B_{j_l})$", the third step represents synchronizations with a single process, which have to be declared explicitly in Atlantif.

It should be noted that the synchronizer translation given in this chapter could be improved, in order to obtain more concise synchronizers without changing the dynamic semantics. For instance, the "**or** 1 **among** $(B_{j_1}, \ldots, B_{j_l})$" of the first step could obviously be deleted if the gate G does not occur in one of the $B_{j_1}, \ldots, B_{j_l}$, or else the list can be shortened to those units where G actually occurs. Similarly, the list "$(B_1, \ldots, B_n)$" in the second step could be reduced to those units where the gate H actually occurs.

Synchronizers produced by such simplifications are not only more concise, but they also indicate *where* synchronizers appear, thus they can provide an easier and more intuitive reading of the code.

Illustration. The example given in Section 4.5.1 stated that a generalized parallel composition operator "**par** *G*#2, *G*#3 **in** *B1* || *B2* || *B3* **end par**" can be expressed in ATLANTIF by a synchronizer "**sync** *G* **is** 2 **or** 3 **among** (*B1*, *B2*, *B3*) **end sync**".

By the dynamic semantics definition of synchronizers (cf. Section 4.5.3), we have

sync(2 **or** 3 **among** (*B1*, *B2*, *B3*)) $= \{\{B1, B2\}, \{B1, B3\}, \{B2, B3\}, \{B1, B2, B3\}\}$
Thus, precisely the same synchronizations as in the original code are possible. Moreover, it can be seen that the ATLANTIF code is as concise as the LOTOS NT code, while providing, in our opinion, a more intuitive notation that is easier to read and to write.

A more complex example of how GPC constructs can be represented in ATLANTIF is given in Appendix B.1 in form of a modelling of the *Open Distributed Processes* protocol.

5.6 Asynchronous termination

5.6.1 Statement

With *asynchronous termination*, we refer to an event in a parallel composition where one of the composed processes is stopped by other processes i.e., the stopped process does not contribute to trigger the stopping event. This opposes the concept of synchronous termination, where a process stops itself. We expressed the latter behaviour e.g., in the example of Section 5.2.2, where *Main_Unit* stops itself by the synchronizer *In* and *Direction_1*, *Direction_2* stop themselves by the synchronizer *CP*.

In LOTOS, asynchronous termination is expressed by the operator "[>". As we wish to show here a combination of asynchronous termination and real time, the following example is written in the ET-LOTOS language (where, for clarity, the delay expression "Δ^x" of ET-LOTOS will be written "**wait** (x)"):

This code describes two processes *P1* and *P2* running in parallel, which begin by waiting at least three and five time units respectively, then synchronize on G, then restart. These two parallel processes may loop arbitrarily often, until the action *RedButton* occurs, which is possible at any moment.

```
(P1[G] || P2[G]) [> RedButton; stop
where
P1[G] := wait (3) G; P1[G],
P2[G] := wait (5) G; P2[G]
```

Figure 5.8: ET-LOTOS code for an example of asynchronous termination

5.6.2 Translation to ATLANTIF

In ATLANTIF, asynchronous termination is expressed by a synchronizer G with at least one synchronization set $\mathcal{U} \in sync(G)$ that is not a superset of the synchronizer's "**stop**-set" $stop(G)$. This can be seen in the translation shown in Fig. 5.9.

```
module Asynchronous_Termination is
dense time
sync G is U1 and U2 end sync
sync RedButton is Supervisor stop U1, U2 end sync
init U1, U2, Supervisor
unit U1 is
  from S1
    wait 3; G; to S1
end unit
unit U2 is
  from S2
    wait 5; G; to S2
end unit
unit Supervisor is
  from S3
    RedButton; stop
end unit
end module
```

Figure 5.9: ATLANTIF translation for the ET-LOTOS code of Fig. 5.8

The processes *P1* and *P2* are represented here by the units *U1* and *U2* respectively, and the behaviour "*RedButton*; **stop**" is represented by the unit *Supervisor*. Note that the latter unit is active from the beginning, otherwise the synchronization by *RedButton* would not be possible. As in former examples, we can notice that the syntax structure is very similar to the initial code.

Correctness of the translation. Schematically, the semantics modulo time additivity (which is respected by the ET-LOTOS semantics rules) of the ET-LOTOS code is presented in Fig. 5.10.

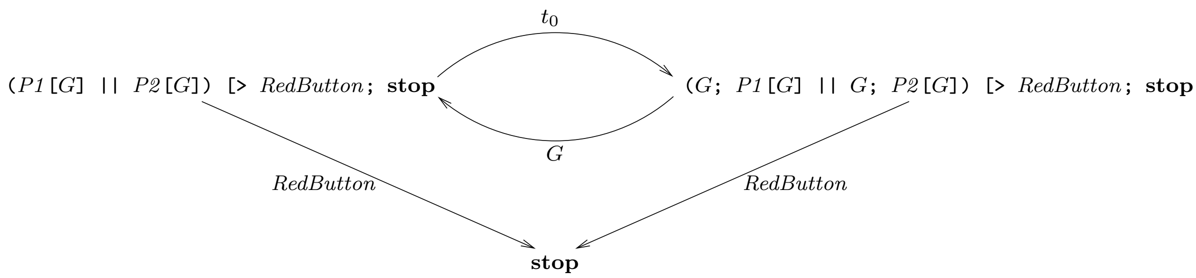

Figure 5.10: Schema of the semantics of the ET-LOTOS example

In this schema, t_0 represents any time value such that $t_0 \geq 5$.

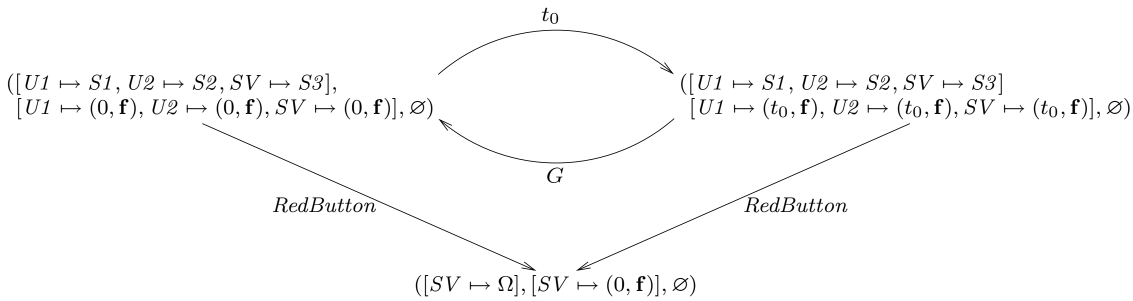

Figure 5.11: Schema of the semantics of the ATLANTIF code of Fig. 5.9

The translation to ATLANTIF has a schematized sematics as given in Fig. 5.11 (where SV corresponds to the unit *Supervisor*). Clearly, the behaviour is not only the same, but the states reached also express in both cases that the processes *P1* and *P2* (or the units *U1* and *U2* respectively) are terminated.

5.7 Exception handling

5.7.1 Statement

A few high-level languages provide constructs to express exception handling i.e., the interruption of a behaviour, followed by the execution of another behaviour, called the exception handler.

Simple exception handling. In LOTOS NT, an exception is raised using the **raise** statement, and exception handling is described in a **trap** statement. A simple example is given in the fragment of LOTOS NT code of Fig. 5.12, where x is an integer variable, G a gate, and B an arbitrary behaviour.

This code begins by checking if the variable x is between 2 and 10, in which case it is then reduced by 1. Otherwise, the exception "*EX1*" is raised, and parametrized by x.

```
trap exception EX1 is z: nat -> G!z in
  if ((x > 1) and (x < 11)) then
    x := x - 1
  else
    raise EX1!x
  end if
end trap;
B
```

Figure 5.12: LOTOS NT code representing simple exception handling

The exception handler consists simply of a gate communication by G, with an emission offer having the parameter's value. Both the regular behaviour and the exception handler are followed by B.

Note that raising an exception in LOTOS NT does not produce a discrete transition, but a silent control passage to the exception handler.

Exception handling with sequential and parallel composition. When the scope of a trap only covers sequential behaviour as above, its translation is relatively simple. Therefore, we also discuss a more complex example, which also integrates parallel composition and has the LOTOS NT code shown in Fig. 5.13, where x and y are integer variables, G a gate, and B an arbitrary behaviour.

Within the scope of the **trap**, we now have almost the same code as above, but it is followed (by sequential composition) by two other **if** statements, which are composed in parallel.

5.7.2 Translation to ATLANTIF

Simple exception handling. In Fig. 5.14, we show those fragments of an ATLANTIF module that correspond to the LOTOS NT code of Fig. 5.12.

The normal behaviour is represented by *Main_Unit*, the exception handler is represented by the unit *EH*. We require that the units *EH* and *Main_Unit* are always started at the same time, and that the variable x is defined in a superunit of *Main_Unit*. We suppose *Unit_B* to be a translation of the behavior B.

When *Main_Unit* becomes active (and *EH* with it), it performs the same check on x as the LOTOS NT fragment: If x is indeed between 2 and 10, then it is reduced by 1 and *Main_Unit* synchronizes with *EH* on *Normal_Termination*. This synchronizer is an auxiliary construct representing the end of the exception's scope (the **end trap** in LOTOS NT). As **end trap** does not induce a discrete transition, *Normal_Termination* is labelled **silent**. Technically, *Normal_Termination* implements the control passage from *EH* and *Main_Unit* to *Unit_B*.

```
trap exception EX1 is z: nat -> G!z in
  if (x > 1) then
    x := x - 1
  else
    raise EX1!x
  end if;
  par
    if (x < 11) then
      x := x - 1
    else
      raise EX1!x
    end if
   ||
    if (y < 11) then
      y := y - 1
    else
      raise EX1!y
    end if
  end par
end trap;
B
```

Figure 5.13: LOTOS NT code representing complex exception handling

```
...
sync EX1 :silent is EH and Main_Unit stop Main_Unit end sync
sync Normal_Termination:silent is EH and Main_Unit
  stop EH , Main_Unit start Unit_B end sync
sync Termination_After_Exception:silent is EH stop EH start Unit_B end sync
sync G is EH end sync
...
unit EH1 is
  from S1
    select
      EX1 ?z; to S2 [] Normal_Termination; stop
    end select
  from S2
    G!z; to S3
  from S3
    Termination_After_Exception; stop
end unit
unit Main_Unit is
  from S4
    if ((x > 1) and (x < 11)) then
      x := x - 1; Normal_Termination; stop
    else
      EX1 !x; stop
    end if
end unit
...
```

Figure 5.14: ATLANTIF code representing simple exception handling

Raising an exception is represented by a synchronization on *EX1*. As stated above, such a synchronization has to be **silent**.

In unit *EH*, it can be seen that the discrete state *S1* represents a stand-by state for an exception to happen, or otherwise for being stopped by *Normal_Termination*. The occurrence of an exception (i.e., a synchronization on *EX1*) stops *Main_Unit* and sets *EH* to the discrete state *S2*, which represents the original exception handler itself, containing the synchronization on G. The synchronization on *Termination_After_Exception* represents the termination of the exception handler.

In LOTOS NT, of course, it is possible that several raises of one exception are within one "**trap**" environment. Our ATLANTIF code could easily be modified for such a case, without modification of unit *EH* or the synchronizers.

Correctness of the translation. For the sake of simplicity, we suppose the specification to be untimed, because the functioning of exceptions is not directly depending on real-time aspects.

Depending on the value v of the variable x, the LOTOS NT code only describes one of the two following behaviours:

- "x `:=` v `- 1;`B" if $1 < v < 11$
- "G `!`v`;`B" otherwise

In ATLANTIF, from the global state $S \stackrel{\text{def}}{=} ([\mathit{Main\_Unit} \mapsto \mathit{S4}, \mathit{EH} \mapsto \mathit{S1}], \theta, [x \mapsto v])$, the following is possible:

- The predicate
  $$\mathit{synchronizing}((S, \varnothing), \mathit{Normal\_Termination}, \mathbf{f}, ((([\mathit{Unit\_B} \mapsto \mathit{S5}], \theta', \varnothing), \{\{\mathit{Unit\_B}\}\}))$$
  is **true** if $1 < v < 11$ (we suppose *S5* to be the first discrete state of *Unit_B*). Thus, it represents an incomplete synchronization chain (cf. Section 4.6.3) that could continue in *Unit_B*.
- A discrete transition $S \xrightarrow{G\ v} S'$ with $S' = ([\mathit{EH} \mapsto \mathit{S3}], \theta'', [z \mapsto v])$ can be taken if $v \leq 1$ or $v \geq 11$. Similar as above, we then can construct an incomplete synchronization chain beginning with
  $$\mathit{synchronizing}((S', \varnothing), \mathit{Termination\_After\_Exception}, \mathbf{f}, ((([\mathit{Unit\_B} \mapsto \mathit{S5}], \theta', \varnothing), \{\{\mathit{Unit\_B}\}\}))$$
  and continuing in *Unit_B*.

$\theta, \theta', \theta''$ do not need to be detailed, as the specification is untimed. Clearly, these two semantics correspond to the two behaviours of the LOTOS NT code, thus the translation is correct.

Note also that by the synchronizer *EX1*, the value v has been transferred from *Main_Unit* to *EH*. This is a simple way to represent raising an exception with parameters.

Exception handling with sequential and parallel composition. In Sections 5.2.2 and 5.6.2 we already saw that it is inevitable that ATLANTIF code becomes complex when it is obtained by a translation from a combination of parallel and sequential code. The ATLANTIF module shown in Fig. 5.15 is a possible translation of the LOTOS NT code fragment of Fig. 5.13.

Note that the static semantics rules for variable scopes (cf. Section 4.4.4) hold in this example. We have $accessible(x) \supseteq \{Main\_Unit, Parallel\_Unit\_1\}$ and $accessible(y) \supseteq \{Main\_Unit, Parallel\_Unit\_2\}$ (where we suppose x, y to be declared in a superunit of $Main\_Unit$).

Note also that this example is a case of a **stop** list in a synchronizer such that it is not possible for all occurring elements to be stopped at the same time, because it is clearly not possible for *Main_Unit*, *Parallel_Unit_1*, and *Parallel_Unit_2* to be active at the same time.

5.8 Lossy buffer

5.8.1 Statement

A lossy buffer is a communication medium between processes that transmits messages which can get lost. Such media being a common problem encountered in formal modelling and verification, lossy buffers often occur as examples in the literature, in many different variations. For this reason, the IF-2.0 model even provides one (derived) syntax construct to express lossy buffers.

We will present here one variation of a lossy buffer expressed in the ET-LOTOS language. It is a simplified model with a capacity of only one message at a time.

Again, Fig. 5.16 gives only a fragment of the code which is thus restricted to the lossy buffer itself (process *LB*) and the parallel composition of the different processes. The processes *Producer_1* to *Producer_5* and *Consumer_1* to *Consumer_4* are not detailed here; we only have the information that *LB* can synchronize on gate *In* with exactly one of the producers and on gate *Out* with exactly one of the consumers.

The process *LB* describes an infinite loop that always begins with a synchronization on gate *In* while receiving a data package by an input offer. After waiting five time units (the minimal transmission time), it can then either synchronize within fewer than five more time units on gate *Out*, emitting the data package by an output offer, or, if this limit is reached, it signals by an internal action **i** that the data package is lost.

5.8.2 Translation to ATLANTIF

As in former examples, Fig. 5.17 gives only a fragment of the ATLANTIF code obtained by translation of the ET-LOTOS fragment of Fig. 5.16. More precisely, we describe the synchronizers that express parallel composition as well as the unit that corresponds to

```
...
sync EX1:silent is EH and (Main_Unit or Parallel_Unit_1 or Parallel_Unit_2)
  stop Main_Unit, Parallel_Unit_1, Parallel_Unit_2 end sync
sync Control_Passage:silent is Main_Unit stop Main_Unit
  start Parallel_Unit_1, Parallel_Unit_2 end sync
sync Normal_Termination:silent is EH and Parallel_Unit_1 and Parallel_Unit_2
  stop EH, Parallel_Unit_1, Parallel_Unit_2 start Unit_B end sync
sync Termination_After_Exception:silent is EH stop EH start Unit_B end sync
sync G is EH end sync
...
unit EH1 is
  from S1
    select
      EX1 ?z; to S2 [] Normal_Termination; stop
    end select
  from S2
    G!z; to S3
  from S3
    Termination_After_Exception; stop
end unit
unit Main_Unit is
  from S4
    if ((x > 1) and (x < 11)) then
      x := x - 1; Control_Passage; stop
    else
      EX1 !x; stop
    end if
  unit Parallel_Unit_1 is
    from S5
      if (x < 11) then
        x := x - 1; Normal_Termination; stop
      else
        EX1 !x; stop
      end if
  end unit
  unit Parallel_Unit_2 is
    from S6
      if (y < 11) then
        y := y - 1; Normal_Termination; stop
      else
        EX1 !y; stop
      end if
  end unit
end unit -- Main_Unit
...
```

Figure 5.15: ATLANTIF code representing complex exception handling

```
(Producer_1 [In] ||| Producer_2 [In] ||| Producer_3 [In] ||| Producer_4 [In] |||
                                                            Producer_5 [In] )
            |[In]|
          LB [In, Out]
            |[Out]|
(Consumer_1 [Out] ||| Consumer_2 [Out] ||| Consumer_3 [Out] |||
                                                            Consumer_4 [Out] )
where
LB [In, Out]  := In ?data; wait (5)
       (Out !data @t[t < 5]; LB [In, Out]
             □
       wait (5) i; LB [In, Out])
...
```

Figure 5.16: ET-LOTOS code representing a lossy buffer

the process *LB*. As in Section 5.2, we suppose that a type *package* has been defined.

Again, this example shows that complex parallel composition, especially when including multiway synchronization, can be expressed in a simple and intuitive way, with the synchronization formulas of the synchronizers *In* and *Out*.

The unit *LB* translates the process of the same name and keeps a very similar syntax. The most notable difference is that we have to introduce an urgent synchronizer *Lost* to represent the internal action **i**.

Correctness of the translation. On an intuitive level, the correctness follows immediately from the fact that the important constructs (nondeterministic choice and **select** action) are used in practically the same way in both encodings. Technically, correctness can be shown as for the preceding examples.

```
...
sync In is
  LB and 1 among (Producer_1 , Producer_2 , Producer_3 , Producer_4 , Producer_5)
end sync
sync Out is
  LB and 1 among (Consumer_1 , Consumer_2 , Consumer_3 , Consumer_4)
end sync
sync Lost : urgent is LB end sync
...
unit LB is
  variables data : package
  from Ready
    In ?data ;to Transporting
  from Transporting
    wait 5;
    select Out !data in [0,5[ [] Lost in [5, ...[ end select;
    to Ready
end unit
...
```

Figure 5.17: Atlantif code representing a lossy buffer

Chapter 6

Translating ATLANTIF to graphical models

Abstract This chapter relates the Atlantif language to graphical models, by defining translators from subsets of Atlantif to the timed automata dialect taken as input by the tool Uppaal and to the time Petri net dialect taken as input by the tool Tina. These translators enable simulation and formal verification of Atlantif specifications. We also describe more succinctly a translator from untimed Atlantif to the Fiacre language and a tool implementation of those three translators.

6.1 Timed automata

6.1.1 Motivation and principles

Choosing UPPAAL as target model

As described in Section 3.2.2, several tools perform simulation and/or formal verification on timed automata, each of them using a particular dialect of the TA model. Among these possible targets, we chose to translate from Atlantif to the TA dialect of the Uppaal tool [89] for several reasons:

- The TA dialect used in Uppaal has a rich expressive power. For instance, it allows user-defined variable types and functions, distinguishes between local and global variables, and enables not only binary but also broadcast synchronization. The translator makes use of some of these features, others could be useful in future extensions of the translator.

  Furthermore, the syntax features of Uppaal are well-documented and described with semi-formal semantics. This is essential to make a translation possible. A mostly undocumented TA dialect such as that used in the Red tool [123] would not be suitable for our purpose.

- Uppaal is a popular, widely-known tool, which is an important condition to ensure tool maintenance in the future. Several tools taking timed automata as input that were created during the last few years are no longer maintained (e.g., Kronos [126], Rabbit [27]). Maintenance of a tool not only covers immediately useful aspects such as bugfixes and porting to up-to-date operating systems, but also can include extensions of the used TA dialect, which may be used for a wider or more efficient translator in the future.

However, a translator of Atlantif to other TA-based tools is possible in principle and should not be too different from the translator to Uppaal, if the other tool also uses the binary synchronization approach.

The UPPAAL TA dialect

The model used by Uppaal is a network of timed automata as defined in Section 3.2.2. Among the three approaches for the semantics of communications, it implements the binary synchronization approach. In this section, a detailed definition is given, in order to fix a terminology and to point out particularities of the Uppaal dialect.

An Uppaal network of timed automata is defined as a 3-tuple $(\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}, \mathcal{V}_g, \mathcal{C})$, where:

- $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are timed automata, defined below.
- $\mathcal{V}_g$ is a set of global variables, each of which is defined by an identifier, a type, and optionally an initial value. Clocks are also understood as variables and therefore $\mathcal{V}_g$ also contains those clocks that can be accessed by all automata.
- $\mathcal{C}$ is a set of *channels*, which induces the possible transition labels: Given a channel $c \in \mathcal{C}$, "c!" denotes an emission on c and "c?" a reception on c. Optionally, a channel can be declared *urgent*, meaning that time elapsing blocks when a synchronization by this channel is possible. Independently, it can optionally be declared as a *broadcast* channel, meaning that instead of binary synchronization, one emission synchronizes with a maximal number (possibly zero) of receptions.

An Uppaal timed automaton $\mathcal{A}$ is defined by the following components:

- In addition to the global variables of the network (accessible in all TA), there is also a set of local variables $\mathcal{V}_{\mathcal{A}}$, declared in the same way as above. Note in particular that each clock variable can be declared either locally or globally.
- The *locations* of a TA are quadruples of the form (l, s, F, tag), where l is a unique location identifier, s is the location's name (not necessarily unique), F is an invariant formula, and the tag may be "*normal*", "*urgent*" or "*commited*"; exactly one location is declared to be the initial location. If the current location of one TA is an

"urgent" one, then time cannot elapse in this TA (and thus in the whole network). The "commited" tag behaves likewise, with the additional constraint that a discrete transition must be taken in the corresponding TA.

- The transitions have the form (l, F, C, Z, l'), where:
  - l is the origin location, l' the target location (as in the standard case).
  - F is the *guard* formula, defined as a conjunction of atomic formulas, where atomic fomulas are those of Definition 3.8, plus arbitrary boolean conditions on non-clock variables.
  - C is either empty (corresponding to an internal τ-transition) or a label as in Definition 3.8.
  - Instead of a set of clocks to be reset (as in the standard case), Z is a list of assignments on variables. They are executed in their given order when the transition is taken. In particular, they can include the assignments of integer values to clocks, and clock resets being expressed by assigning zero. We note $Z = (a_1, \ldots, a_n)$ for these assignments, and we will use the following notation for the concatenation of two such lists: $((a_1, \ldots, a_n), (a'_1, \ldots, a'_m)) \stackrel{\text{def}}{=} (a_1, \ldots, a_n, a', \ldots, a'_m)$.

Uppaal provides the predefined types "**int**" and "**bool**", as well as arrays and records. It also provides functions in a C-like syntax.

Problems to overcome

The following paragraphs list the three most important aspects of the Atlantif constructs for which no direct translation to Uppaal exists. Instead, emulations had to be implemented. These issues concern aspects related to the Atlantif concurrent structures; the real-time constructs did not raise such problems.

Multiway synchronization. As described in Section 4.5.1, Atlantif allows synchronization of one, two, or more than two processes at the same time. A synchronization on a standard Uppaal channel however only features two participating processes. In the case of a single process, a synchronizer invocation can be translated by using a **broadcast** channel, which can be invoked by one emission without a counterpart. For more than two processes, on the other hand, the usage of broadcast channels is not suitable, because such channels do not ensure that all processes we wish to synchronize actually participate in the synchronization (except for the one that is designated as emission process).

Therefore, it will be necessary to emulate the multiway synchronization by a succession of binary synchronizations. A refinement of this idea is described in Section 6.1.4 on page 145.

Unit stopping and starting. In a network of timed automata, each TA is "running" from the beginning and cannot be stopped. Atlantif units can be started and stopped, and therefore excluded from participating in synchronizations. Therefore, timed automata representing Atlantif units will be defined with an additional auxiliary location that represents inactivity and from which only synchronizations that start this unit are possible.

Offers. The channels of Uppaal do not have communication offers. Atlantif communications using offers must therefore transfer data in another way, using global variables.

Intuition of the translation approach

Each unit u of an Atlantif module is mapped to a single TA, with each discrete state of u being mapped to a TA location and an invariant being synthesized from the **must** constraints of state's action A.

A itself is decomposed into at least one TA transition for each branch of control. The emulation of concurrent and multiway synchronization can induce multiplications of the resulting TA transition. Data exchanges are emulated using TA shared variables. Starting and stopping of units induces additional transitions.

In Section 6.1.3, this translation is defined formally.

6.1.2 Restrictions

Our Atlantif to Uppaal translator only works for a subset of Atlantif. This is for two reasons:

- The expressive power of timed automata in general or of the Uppaal dialect in particular is not sufficient for certain Atlantif constructs.
- The tool only is a prototype, and therefore it does not cover everything. In particular, some translation problems have an obvious solution that, however, complicates the formal definitions; thus they are left out for the sake of readability.

General restrictions

- Only modules with **dense** time domain are translated, because timed automata use dense time. Discrete time behaves differently in some cases, so cannot be translated. Translating an untimed Atlantif module would however be possible – simply by a network of "timed" automata without clocks.
- Function and type definitions are not translated, and only variables of type **int** and **bool** are translated. This restriction is not necessary, as Uppaal has a concept of user-defined types and functions similar to Atlantif. Within the framework of the

syntax that Uppaal offers for these constructs, a translation seems possible (and easy).

Actions

- Non-deterministic assignments are not supported, because in general they would induce an infinite number of transitions, which is not feasible. A translation would be possible if the variable type(s) and/or the condition limited the number of possible assigments to be finite.

- Simultaneous assignments to strictly more than one variable are not supported, although they could be emulated by a succession of simple assignments, using auxiliary variables.

- Each timed expression E (i.e., E occurring in a time window or as the parameter of a **wait** action) must be an integer constant.

  It is necessarily constant, because model-checking on timed automata depends on *clock regions* that have to be determined statically. For this static evaluation, a TA must not contain other variables than clocks in guard and invariant formulas. Timed expressions in an Atlantif module however will obviously be translated to such guard and invariant formulas, thus they must be constants.

  It could be non-integer, although Uppaal also only accepts integer constants in guard and invariant formulas. But as each float value is a rational number, it is possible to redefine the duration of a time unit in order to obtain integer constants[16].

- Time windows may only be intervals. A translation of intersections ("**and**") would be possible without problem, but given the restriction to constant timed expressions, the usage of conjunctions makes no sense (because the intersection of two intervals could be expressed by a single interval in the first place). A translation of unions ("**or**") would be possible, using the decompositions described in Proposition 4.6 on page 97. It is necessary to refer to such an approach, as Uppaal only allows guard formulas that are conjunctions.

- An execution path of a multibranch transition must not contain a condition expression E (i.e., the boolean expression that is evaluated in an **if** action) such that $use(E)$ contains a variable V that occurs as a reception offer "?V" earlier on this execution path. In Uppaal, conditions on a transition are always evaluated before the assignments of this transition, thus the translation of a condition has to anticipate earlier assignments statically. Such an anticipation is possible for a deterministic assignment action (implemented by the function *update_exp*, cf. page 140), but not for a value reception, which is not deterministic in general.

[16] For instance, if a module contains two timed expressions "$\frac{1}{3}$" and "0.25", we could replace them by the integer expressions "4" and "3" respectively. As this means refining our time units by a factor of twelve, possible timed expressions in formulas of timed temporal logic we wish to verify by model checking also have to be multiplied by twelve.

A future version of the translator could lift this restriction by dividing such paths into two consecutive transitions.

- **while** loops are not translated, although this would not make theoretical difficulties (cf. Section 6.1.4 on page 144).

- **case** actions are not translated, although again this would not make theoretical difficulties (translations are described e.g., in [114]). Note also that **if** actions are translated, those being a special case of **case** actions.

- An execution path containing a communication action that uses an **urgent** or a **silent** synchronizer must not be time-constrained i.e., must not contain any **wait** actions, and only time windows of the form "`[0,...[`". This is because otherwise, the translation would require clock constraints on transitions labelled with urgent channels, which are prohibited in UPPAAL. Although in some cases one could use invariants instead of clock constraints on transitions, a general solution to lift this restriction has not been found.

- Offers have to have one of the forms "`?`V", "`?`**any** T", "`!`E". The translation of offers with constructor patterns is probably difficult, but possible when user-defined types of ATLANTIF are translated to UPPAAL (see above). A translation of offers of the form "`?(`P **where** E`)`" seems also possible by adding constraints in the guard formula.

- Variable assignments are not translated if they are (within an execution path) behind a communication on a gate G such that $stop(G)$ contains the current unit. This restriction is due to the technical definition of the translator; but in practice, assigning a variable after a "stopping synchronization" does not seem reasonable anyway.

- Neither the **then** nor the **else** branch of an **if** action may contain a communication action with a **must** time window. In timed automata, invariants in locations are used to represent strong deadlines. Such invariants apply always, independently of the satisfaction of data conditions of outgoing transitions. Therefore, it is a non-trivial problem to generate a single transition with a data condition and a strong deadline, a solution for which has not been found.

Synchronizers

- For each synchronizer G, all offer lists of communications on G must match in type and cardinality. In our translator, offer synchronization is emulated by variable assignments with a fixed number of variables of fixed types. Extending our translator to accept different offer profiles on a single sychronizer could be possible, by defining separate channels for each offer profile.

- For each synchronizer G, for each synchronization set $\mathcal{U} \in sync(G)$, there must be exactly one unit $u \in \mathcal{U}$ such that:

– In each communication by G occurring in u, all offers are emissions.
– For each $u' \in (\mathcal{U} \setminus \{u\})$, in each communication by G occurring in u', all offers are receptions.

This condition ensures that in each synchronization that can be executed, exactly one communication action only emits offers, and all the other communication actions only receive offers. Such a restriction is necessary to exclude three constructs that are not supported: (1) synchronization of two or several emission offers, (2) synchronization of reception offers without emission (i.e., a value would have to be generated, which corresponds to a non-deterministic assignment), and (3) a mix of reception and emission offers within the offer list of one communication action.

6.1.3 Definition of the translator

We suppose an Atlantif module M that satisfies the static semantics constraints of Chapter 4 and the restrictions listed in Section 6.1.2. As in Chapter 4, we note $\mathbb{G}$ for the set of synchronizers/gates, $\mathcal{V}$ for the set of variables, and $\mathbb{U}$ for the set of units.

Translation of one module into a network of TA

The network of timed automata obtained from the module M is defined as follows:

- For each $u \in \mathbb{U}$, one TA $\mathcal{A}_u$ is constructed as defined below (page 136).
- The set $\mathcal{V}_g$ of global variables is the disjoint union of two subsets. The first subset contains one element for each variable in $\mathcal{V}$ (i.e., from all units of $\mathbb{U}$), using the same identifier, the same type, and (if applicable) the same initial value[17].

  The second subset contains auxiliary variables used for the emulation of offers: For each gate $G \in \mathbb{G}$ that exchanges $n > 0$ offers in each of its synchronizations, we define n variables written $G\_aux\_1, \ldots, G\_aux\_n$.
- The set $\mathcal{C}$ of channels is defined as described in the following paragraph.

Synchronizers. Each synchronizer $G \in \mathbb{G}$ translates to several channels, as follows:

- Let n be the number of unit sets in $sync(G)$ ($n = card(sync(G))$). Then γ_G is defined as a bijective function from the set $1..n$ into $sync(G)$ (cf. page 78).
- For each $i \in 1..n$, one unit u_0 is chosen from the set $\gamma_G(i)$, where the following is satisfied:

[17]This means that all variables get assigned their initial value, regardless of their unit being initial or not. Given that in Uppaal, variables always have a value anyway (arbitrary, if no assignment has yet occurred), this is not a problem.

- If gate G uses offers, then u_0 is the single unit in $\gamma_G(i)$ where the offers are emissions (as stated among the restrictions of Section 6.1.2).
- Otherwise, u_0 is chosen arbitrarily[18].

Then, for given G and i, we define the predicate $\mathit{emission}(G, i, u)$ to be true if and only if $u = u_0$.

We define the function $\mathit{sense}(G, i, u) : (\mathbb{G} \times \mathbb{N} \times \mathbb{U}) \rightarrow \{!, ?\}$ as follows

$$\mathit{sense}(G, i, u) \stackrel{\text{def}}{=} \begin{cases} ! & \text{if } \mathit{emission}(G, i, u) \\ ? & \text{otherwise} \end{cases}$$

This function will be needed in the translation of actions.

- Let m be the number of units in $\gamma_G(i)$ ($m = \mathit{card}(\gamma_G(i))$). Then γ_G^i is defined as another bijective function, from the set $\{1, \ldots, m-1\}$ into $(\gamma_G(i) \setminus \{u_0\})$. Depending on m, the synchronization set $\gamma_G(i)$ translates to the following channels:

  - If $m = 1$ (i.e., $\gamma_G(i) = \{u_0\}$), then one **broadcast** channel named G_i_1 is defined (declared urgent in UPPAAL if G is urgent or silent in ATLANTIF).
  - If $m = 2$, then one normal channel named G_i_1 is defined (declared urgent if G is urgent or silent).
  - If $m > 2$, then $(m-1)$ normal channels named $G\_i\_1, \ldots, G\_i\_(m-1)$ are defined (declared urgent if G is urgent or silent).

  Intuitively, a channel "G_i_j" thus represents the jth step of a synchronization by the ith synchronization set of G.

- If the set $\mathit{stops}(G)$ is not empty, then one **broadcast** channel named G_stops is defined. The translation of the units in the set $\mathit{stops}(G)$ will be "stopped" by each execution of this broadcast.

- If the set $\mathit{starts}(G)$ is not empty, then one **broadcast** channel named G_starts is defined. The translation of the units in the set $\mathit{starts}(G)$ will be "started" by each execution of this broadcast.

Note that we consider **silent** synchronizers exactly like **urgent** synchronizers i.e., they both translate to (UPPAAL-)urgent channels. This lack of precision is discussed in Section 6.1.4.

Translation of one unit into one timed automaton

A unit $u \in \mathbb{U}$ translates to a timed automaton $\mathcal{A}_u$ as follows:

[18] In the tool implementation, u_0 is chosen as the unit with the fewest occurrences of G; this choice favors a smaller size of the translation.

- The set $\mathcal{V}_{\mathcal{A}_u}$ of local variables only contains one element "*CLOCK_u*" of type clock.
- The set of transitions and the set of locations are both defined by the function *transitions_and_locations*(u), described below. Among these locations, the initial location is either l_1 (i.e., the location corresponding to the first discrete state of u) if u is an initial unit of the module M, or l_d (i.e., the location corresponding to Ω) otherwise.

Transitions and locations. The construction of transitions and locations is based on the set of *pre-transitions* defined by u, each of which represents one execution path (cf. page 72) in one of the multibranch transitions of u. A pre-transition has the form $(s, F, C, Z, s', \Delta, \lambda)$ and corresponds to an execution path as follows:

- The path originates from the discrete state s and terminates with a jump to the discrete state s' (if the path terminates with a **stop** action, then $s' = \Omega$).
- The path can only be executed if the boolean condition F is satisfied.
- The communication of the path translates to a label C (containing information on the gate used, the synchronization set used, and whether it is an emission or a reception); if the path does not contain a communication, then C is empty.
- The path induces the variable assignments Z.
- The **wait** actions occurring on the path (if any) describe a total delay of Δ.
- If the path contains a communication with a "**must**" time window, then its upper limit equals λ, otherwise $\lambda = \infty$.

The construction of these pre-transitions is performed by the function *trans* defined page 139.

Given the pre-transitions, the function *transitions_and_locations* calculates all transitions and locations for the translation of a unit u. It is defined by the pseudocode given in Fig. 6.1.

Intuitively, it creates two sets L (locations, defined page 130) and T (transitions, defined page 131) in three steps:

- First, L is initialized to contain one location for each discrete state s, and one location named "disabled", representing the state Ω. T is initialized empty.
- Second, transitions are defined for asynchronous termination (corresponding to cases, where u is stopped by a synchronization in which it does not communicate) and for starting by synchronizers. These are implemented by reception labels of the form "*G_stops*?" and "*G_starts*?" respectively. No new locations are defined in this step.

$\mathit{transitions\_and\_locations}(u) \stackrel{\text{def}}{=}$
let $s_1, \dots, s_n$ be the discrete states of u
$P \leftarrow \bigcup_{i \in 1..n} \mathit{trans}(\mathit{act}(s_i), (s_i, \mathbf{true}, \varnothing, (), \delta, 0, \infty))$
$L \leftarrow \{(l_1, s_1, \mathbf{true}, \text{normal}), \dots, (l_n, s_n, \mathbf{true}, \text{normal}),$ ($l_1, \dots, l_n, l_d$ chosen pairwise distinct)
    $(l_d, \text{“disabled”}, \mathbf{true}, \text{normal})\}$
$T \leftarrow \varnothing$
for each $G \in \mathbb{G}$
  if $(\exists \mathcal{U} \in \mathit{sync}(G))\ u \notin \mathcal{U}$ then (*only asynchronous termination here*)
    if $u \in \mathit{stop}(G)$ then
      $T \leftarrow T \cup \{(l, \mathbf{true}, G\_\mathit{stops}?, (), l_d) \mid (l, \cdot, \cdot, \cdot) \in L)\}$
    end if
    if $u \in \mathit{start}(G)$ then
      $Z_u^i \leftarrow$ assignments for initial variable values of u
      $T \leftarrow T \cup \{(l_d, \mathbf{true}, G\_\mathit{starts}?, Z_u^i, l_1)\}$
    end if
  end if
end for
loop while $P \neq \varnothing$
  choose $(s, F, C, Z, s', \Delta, \lambda)$ from P (*by construction*: $\Delta = 0$ *or* $s' = \delta$)
  $P \leftarrow P \setminus \{(s, F, C, Z, s', \Delta, \lambda)\}$
  find l, l', F' such that $(l, s, F', \cdot), (l', s', \cdot, \cdot) \in L$
  if $\lambda \neq \infty$ then
    replace $(l, s, F', \cdot)$ in L
        by $(l, s, F' \wedge \mathit{CLOCK\_u} \leq \lambda, \text{normal})$
  end if
  if $s' \neq \delta$ then (*filter paths without target state*)
    if $C = \varnothing$ then
      $T \leftarrow T \cup \{(l, F, \varnothing, \mathit{append}(Z, \mathit{CLOCK\_u} := 0), l')\}$
    else
      let $G, i, \wr$ be such that $C = G\_i\wr$
      if $\wr = ?$ then
        $j \leftarrow (\gamma_G^i)^{-1}(u)$ (*find position among synchronizations*)
        $T \leftarrow T \cup \{(l, F, G\_i\_j?, \mathit{append}(Z, \mathit{CLOCK\_u} := 0), l')\}$
      else
        $m \leftarrow \mathit{max}(1, \mathit{card}(\gamma_G(i)) - 1)$
        $Y \leftarrow (G\_i\_1, \dots, G\_i\_m)$ (Y: *list of channel identifiers*)
        if $\mathit{stop}(G) \setminus \gamma_G(i) \neq \varnothing$ then
          $Y \leftarrow \mathit{append}(Y, G\_\mathit{stop})$
        end if
        if $\mathit{start}(G) \setminus \gamma_G(i) \neq \varnothing$ then
          $Y \leftarrow \mathit{append}(Y, G\_\mathit{start})$
        end if
        $k \leftarrow \mathit{length}(Y)$
        if $k \leq 1$ then (*no multiway syncr. emulation*)
          $T \leftarrow T \cup \{(l, F, G\_i\_1!, \mathit{append}(Z, \mathit{CLK\_u} := 0), l')\}$
        else (*multiway syncr. emulation*)
          $l'_2 \leftarrow \mathit{new\_location}(), \dots, l'_k \leftarrow \mathit{new\_location}()$
          $T \leftarrow T \cup \{(l, F, Y[1]!, Z, l'_2),$ (*containing guard and assignments*)
              $(l'_2, \mathbf{true}, Y[2]!, \varnothing, l'_3), \dots,$
              $(l'_{k-1}, \mathbf{true}, Y[k-1]!, \varnothing, l'_k),$
              $(l'_k, \mathbf{true}, Y[k]!, (\mathit{CLOCK\_u} := 0), l')\}$ (*containing clock reset*)
          $L \leftarrow L \cup \{(l'_2, \mathit{AUX\_G\_i\_2}, \mathbf{true}, \text{commited}), \dots,$
              $(l'_k, \mathit{AUX\_G\_i\_k}, \mathbf{true}, \text{commited})\}$ (*addition of new locations*)
        end if
      end if
    end if
  end if
end loop

Figure 6.1: Pseudocode defining the function *transitions_and_locations*

$$\begin{array}{l}
\mathit{trans}(\mathbf{null}, (s, F, C, Z, \delta, \Delta, \lambda)) \stackrel{\text{def}}{=} \{(s, F, C, Z, \delta, \Delta, \lambda)\} \\
\mathit{trans}(\mathbf{reset}\ V_1\,\mathbf{,}\ \ldots\ \mathbf{,}V_n, (s, F, C, Z, \delta, \Delta, \lambda)) \stackrel{\text{def}}{=} \{(s, F, C, Z, \delta, \Delta, \lambda)\} \\
\mathit{trans}(V\,\mathtt{:=}E, (s, F, C, Z, \delta, \Delta, \lambda)) \stackrel{\text{def}}{=} \{(s, F, C, (Z, (V := E)), \delta, \Delta, \lambda)\} \\
\mathit{trans}(\mathbf{wait}\ c, (s, F, C, Z, \delta, \Delta, \lambda)) \stackrel{\text{def}}{=} \{(s, F, C, Z, \delta, \Delta + c, \lambda)\} \\
\mathit{trans}(G\ O_1 \ldots O_n\ Q\ \mathbf{in}\ W, (s, F, C, Z, \delta, \Delta, \lambda)) \ \stackrel{\text{def}}{=} \\
\quad \begin{cases}
\{(s, F \wedge \mathit{new_guard}(\mathit{CLOCK_u}, W, \Delta, \mathit{new_limit}(\Delta, Q, W)), G_i\wr, & \\
\quad (Z, \mathit{tl_off}(G, O_1 \ldots O_n)), \delta, 0) \mid u \in \gamma_G(i) \wedge \mathit{sense}(G, i, u) = \wr\} & \text{if } u \notin \mathit{stop}(G) \\
\{(s, F \wedge \mathit{new_guard}(\mathit{CLOCK_u}, W, \Delta, \mathit{new_limit}(\Delta, Q)), G_i\wr, & \\
\quad (Z, \mathit{tl_off}(G, O_1 \ldots O_n)), \text{“disabled”}, 0) \mid u \in \gamma_G(i) \wedge \mathit{sense}(G, i, u) = \wr\} & \text{if } u \in (\mathit{stop}(G) \setminus \mathit{start}(G)) \\
\{(s, F \wedge \mathit{new_guard}(\mathit{CLOCK_u}, W, \Delta, \mathit{new_limit}(\Delta, Q)), G_i\wr, & \\
\quad (Z, \mathit{tl_off}(G, O_1 \ldots O_n)), s_1, 0) \mid u \in \gamma_G(i) \wedge \mathit{sense}(G, i, u) = \wr\} & \text{else}
\end{cases} \\
\mathit{trans}(\mathbf{stop}, (s, F, C, Z, \delta, \Delta, \lambda)) \stackrel{\text{def}}{=} \begin{cases}
\varnothing & \text{if } C = \varnothing \\
\{(s, F, C, Z, \text{“disabled”}, 0, \lambda)\} & \text{if } C \neq \varnothing, \Delta = 0 \\
\{(s, (\mathit{CLOCK_u} \geq \Delta) \wedge F, C, Z, \text{“disabled”}, 0, \lambda)\} & \text{otherwise}
\end{cases} \\
\mathit{trans}(\mathbf{to}\ s', (s, F, C, Z, \delta, \Delta, \lambda)) \stackrel{\text{def}}{=} \begin{cases}
\{(s, F, C, Z, s', 0, \lambda)\} & \text{if } \Delta = 0 \\
\{(s, (\mathit{CLOCK_u} \geq \Delta) \wedge F, C, Z, s', 0, \lambda)\} & \text{otherwise}
\end{cases} \\
\mathit{trans}(A_1\,\mathbf{;}\ A_2, (s, F, C, Z, \delta, \Delta, \lambda)) \ \stackrel{\text{def}}{=} \\
\quad (\bigcup_{(s, F', C', Z', \delta, \Delta', \lambda') \in \mathit{trans}(A_1, (s, F, C, Z, \delta, \Delta, \lambda))} \mathit{trans}(A_2, (s, F', C', Z', \delta, \Delta', \lambda'))) \\
\quad\quad \cup \{(s, F', C', Z', s', \Delta', \lambda') \in \mathit{trans}(A_1, (s, F, C, Z, \delta, \Delta, \lambda)) \mid s' \neq \delta\} \\
\mathit{trans}(\mathbf{select}\ A_0\ \mathtt{[]} \ldots \mathtt{[]}\ A_n\ \mathbf{end}, (s, F, C, Z, \delta, \Delta, \lambda)) \ \stackrel{\text{def}}{=} \\
\quad \bigcup_{i \in 0..n} \mathit{trans}(A_i, (s, F, C, Z, \delta, \Delta, \lambda)) \\
\mathit{trans}(\mathbf{if}\ E\ \mathbf{then}\ A_1\ \mathbf{else}\ A_2\ \mathbf{end}, (s, F, C, Z, \delta, \Delta, \lambda)) \ \stackrel{\text{def}}{=} \\
\quad \mathit{trans}(A_1, (s, \mathit{update_exp}(E, Z) \wedge F, C, Z, \delta, \Delta, \lambda)) \cup \\
\quad\quad\quad \mathit{trans}(A_2, (s, (\neg \mathit{update_exp}(E, Z)) \wedge F, C, Z, \delta, \Delta, \lambda))
\end{array}$$

Figure 6.2: The mapping *trans*

- Third, a loop on the set P of pre-transitions is performed. Each pre-transition is either translated into a single transition in T, or is is expanded into a chain of transitions using new auxiliary locations. The latter applies if the two following conditions are both satisfied: (1) The current unit is the emission unit for the current synchronization set, and (2) the synchronization set has more than two elements. The auxiliary function $\mathit{new\_location}() = l$ used in the pseudocode returns an as yet unused location identifier “l”[19]. We will illustrate this expansion in Example 6.1.

Actions. Each action A translates to one or several pre-transitions, formally defined by the mapping *trans* in Fig. 6.2. Technically, *trans* takes as input A and one pre-transition representing an incomplete path. Then, it determines the pre-transition(s) corresponding to this incomplete path followed by A[20]. Note that s_1 refers to the first discrete state of the unit u.

[19]Note that this is of course an informal notation, as *new_location* obviously depends on the identifiers already defined instead of being without parameters

[20]For this reason, *trans* is used in the pseudocode of the function *transitions_and_locations* with the second parameter $(s_i, \mathbf{true}, \varnothing, (), \delta, 0, \infty)$, representing an empty path.

Note that *trans* is defined only for those actions accepted by our translator (cf. Section 6.1.2). In particular, it does not cover nondeterministic assignments and **case** actions.

The function *trans* makes use of the auxiliary functions *tl_off*, *update_exp*, *new_guard*, and *new_limit*, described in the following:

Offers. Offers of the form "$O_1 \ldots O_m$" are translated by the function *tl_off* ("*translate offers*") into a list of assignments. The function also uses the gate identifier G as an input parameter, which is used to determine the auxiliary variables $G\_aux\_1, \ldots, G\_aux\_m$.

$$tl\_off(G, O_1 \ldots O_m) \stackrel{\text{def}}{=} \begin{cases} (G\_aux\_1 := E_1, \ldots, G\_aux\_m := E_m) & \text{if } O_1 \ldots O_m = \,!E_1 \ldots !E_m \\ (V_1 := G\_aux\_1, \ldots, V_m := G\_aux\_m) & \text{if } O_1 \ldots O_m = \,?V_1 \ldots ?V_m \end{cases}$$

Expressions. The translation of expressions is purely syntactical; for instance, ATLANTIF uses the equals sign "=" to express equality, whereas UPPAAL uses the symbol "==". These translations are therefore trivial and thus not detailed here.

ATLANTIF expressions containing variables can be (partially) evaluated with respect to a chain of assignments that concern the variables of the expression. For this evaluation, we define the function *update_exp* as follows:

$$update\_exp(E, (V_1 := E_1, \ldots, V_n := E_n)) \stackrel{\text{def}}{=} [E_1/V_1] \ldots [E_n/V_n]E$$

In this definition, the assignments are applied to E in *inverse* order, because the value of a variable in E_i may be given by an assignment earlier in the list of assignments. For instance, this is obvious with the assignment list $(V := 1, V' := V)$ applied to the expression V'.

Time windows. By the function $new\_guard(X, W, \Delta)$, a time window W (which is a mere interval due to the syntactic restrictions defined in Section 6.1.2) translates to a boolean formula, which is afterwards integrated into the guard of the according pre-transition. This translation also depends on the **wait** actions of the execution path i.e., it depends on the current delay Δ of the incomplete path on which the communication action follows. The parameter X is the clock identifier of the current unit's translation. Formally:

$$new\_guard(X, W, \Delta) \stackrel{\text{def}}{=} \begin{cases} X - \Delta \geq n \wedge X - \Delta \leq m & \text{if } W = [n, m] \\ X - \Delta > n \wedge X - \Delta \leq m & \text{if } W =]n, m] \\ X - \Delta \geq n \wedge X - \Delta < m & \text{if } W = [n, m[\\ X - \Delta > n \wedge X - \Delta < m & \text{if } W =]n, m[\\ X - \Delta \geq n & \text{if } W = [n, \ldots[\\ X - \Delta > n & \text{if } W =]n, \ldots[\end{cases}$$

It can thus be seen that the delay induced by **wait** actions affects the pre-transition by the function *new_guard*, in the evaluation of a communication action. If an execution path does not contain a communication action, then it can be seen from the definition of *trans* that the Δ is evaluated at the end of this execution path i.e., in the case of a **stop** or a **to** action.

Strict time limits in communications. The limit of the time window of a communication action with **must** modality translates to a value $\lambda \in \mathbb{N} \cup \{\infty\}$ within a pre-transition. Again, such a limit also depends on the **wait** actions of the execution path i.e., a value Δ. Formally, this is defined by the function *new_limit* as follows:

$$new\_limit(\Delta, Q, W) \stackrel{\text{def}}{=} \begin{cases} \Delta + m & \text{if } Q = \textbf{must} \text{ and } W \text{ is of the form } [n,m] \text{ or }]n,m] \\ \infty & \text{otherwise} \end{cases}$$

Example 6.1. *This example shows how the decompsition of multiway synchronizations works in practice. Fig. 6.3 contains fragments of an* Atlantif *module with multiway synchronization. We suppose four units that represent processes running in parallel, where one among these processes coordinates the others. Under certain conditions, this unit "Leader_Process" may send an cancel message, along with an error code, to the three other units.*

```
(...)
sync Cancel is Leader_Process and Process1
        and Process2 and Process3 end sync
(...)
unit Leader_Process is
  (...)
  from Processing
    (...)
    Cancel !Error_Code in [10,...[;
    to Recovery
  (...)
end unit
unit Process1 is
  (...)
    Cancel ?Error1
  (...)
end unit
unit Process2 is
  (...)
    Cancel ?Error2
  (...)
end unit
unit Process3 is
  (...)
    Cancel ?Error3
  (...)
end unit
(...)
```

Figure 6.3: Fragments of an Atlantif module with multiway synchronization

Fig. 6.4 shows the fragments of the translated timed automata corresponding to the code of Fig. 6.3, one for each translated unit.

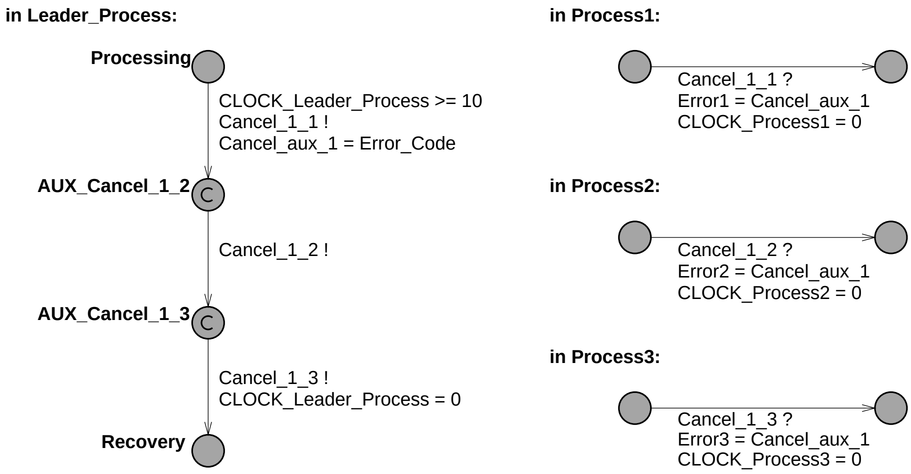

Figure 6.4: Translation fragments in UPPAAL TAs

The unit Leader_Process is chosen such that emission$(Cancel, 1, Leader\_Process)$ *is* **true** *(cf. page 136), because its offer on the gate Cancel is an emission (while the other units only have reception offers). Thus, the emulation of the 4-ary synchronization is realized by a succession of three transitions in the translation of Leader_Process, and by one transition each in the translation of the other units.*

Note that the clock constraint that translates the time window "`[10,...[`*" from unit Leader_Process appears in the resulting TA on the first transition among the three. The two other transitions follow the first transition immediately (without delay), because of the usage of* UPPAAL *commited locations. Therefore, the synchronization chain can be started once the constraint is satisfied, and it is then executed without time elapsing.*

Note also that each *occurrence of "Cancel" in an execution path of the unit Leader_Process induces three transitions and two auxiliary places, as shown in Fig. 6.4.*

The example also illustrates the emulation of communication offers we implemented in our translator: In the first of the three transitions in the translation of Leader_Process, the auxiliary variable Cancel_aux_1 is assigned with the value of Error_Code, which corresponds to the emission offer in the ATLANTIF *module. In each translation of the other units, the variable Errori* $(i \in 1..3)$ *is assigned with the value of Cancel_aux_1, which corresponds to the reception offers in the* ATLANTIF *module.*

The semantics of UPPAAL *define that in a synchronization on a gate G, first the assignments on the transition labelled "G*`!`*" are evaluated, then the assignments on the transition labelled "G*`?`*". Thus, the value of Error_Code is correctly transmitted to the variables* $Error1, Error2, Error3$*: The first synchronization (labelled "Cancel_1_1") executes first the assignment "Cancel_aux_1 := Error_Code", then the assignment "Error1 := Cancel_aux_1". As Error_Code does not change its value during the following two transitions, Error2 and Error3 also receive the correct value.*

6.1.4 Discussion

Impact of the restrictions

The list presented in Section 6.1.2 defining the subset of ATLANTIF that can be translated into timed automata is not as restrictive as its length might suggest. First, several of the restrictions can be avoided by alternative constructs, for instance in the following cases:

- Simultaneous assignments to several variables can always be represented by a succession of assignments to single variables (possibly using auxiliary variables). For instance, the variable swap "V_1, V_2 := V_2, V_1" we mentioned on page 62 can be defined by "V' := V_1; V_1 := V_2; V_2 := V'", where V' is an auxiliary variable.
- Non-integer time constants can be represented by changing the granularity of time units, as described in Section 6.1.2 on page 133.

Besides such merely technical restrictions, actual limitations are caused by those restrictions applying to offers, to urgent synchronizers, and to **must** communications within **if** actions. None of the examples in Chapter 5 (which intends to explore the expressive power of ATLANTIF and provides therefore a broad choice of examples) would be excluded by the restrictions on offers or on **must** communications, but two examples apply timing constraints to urgent synchronizers. In both cases however, the same behaviour could be produced using **must** timing constraints instead of urgency.

Therefore, the impact of the restrictions seems reasonably small.

Imprecision problems of the translator

A more serious problem than the restrictions just discussed are those translations that do not entirely preserve the semantics. This concerns minor technical issues such as not taking notice of variable resets, but also more complex matters, discussed below:

Multiway synchronization emulation. One of the central ideas of the translator from ATLANTIF to UPPAAL is the emulation of multiway synchronizations by a chain of binary synchronizations. Given the restriction of UPPAAL to binary synchronizations, the emulation by such a chain is clearly inevitable, therefore no translation approach could keep the semantics.

However, we consider the impact of this problem as small. By using the "commited" location construct provided by UPPAAL, the chain of n synchronizations always results in a succession of n corresponding discrete transitions in the semantic model, such that it cannot be interrupted by timed transitions nor by discrete transitions with other labels. Therefore, an obvious analogy exists between the behaviour of the ATLANTIF module and the behaviour of its translation.

Note that in some cases the translator may introduce timelocks that do not exist before. A solution for this inconvenience is sketched below on page 145.

Paths without communication action and silent synchronizers. Two similar problems concern transitions labelled with ε (which are eliminated by the ATLANTIF semantics but that occur in the translation's semantics).

The first problem is that our translator produces for each path in a multibranch transition one transition in a timed automaton, independent of whether there is a communication action on this path. The ATLANTIF semantics would normally eliminate paths without communication actions by the rule (ε-*elim*) (cf. page 73).

The second problem is that our translator produces for each synchronization at least one transition in each participating automaton, even if the synchronizer is defined as **silent** (the translator treats **silent** synchronizers like **urgent** synchronizers). But in the ATLANTIF semantics, **silent** synchronizations are eliminated by the predicate *enabled*.

The examples of Chapter 5 indicate that silent synchronizers may occur often when an ATLANTIF module models e.g., control passages, interleavings of parallel and sequential behaviour, or exceptions.

In the general case, it is inevitable for a translator to have these problems: Regarding the first problem, the alternative would be a translator that explores from each discrete state all successions of multibranch transition paths until one contains a communication. But then, the number of transitions could not only explode, but even become infinite if paths without communications are forming a loop. Regarding the second problem, an alternative translator that respects the elimination would have similar trouble with exploding numbers of transitions and loops. Moreover, chains of synchronizations can include several units, thus we still would be obliged to emulate a translated synchronization chain by several binary synchronizations.

Therefore, it seems indicated to use a translation approach like the one we proposed, in spite of the semantic problems. If the ATLANTIF code satisfies certain conditions, these imprecisions have no impact on properties expressed in the *linear time temporal logic* (LTL$_{-X}$): This is the case if taking an ε-transition never resolves a choice. For this, it is sufficient to demand that in each multibranch transitions that contains a path without gate communication or a path with a gate communication by a **silent** synchronizer, no **select** action may occur.

Possible improvements and extensions

Translation of *while* actions. There are at least two possible ways to translate **while** actions:

First, by translating each multibranch transition path containing a **while** action not to a single transition in the resulting TA, but instead to one auxiliary location l (labelled as "commited") and three transitions: One transition from the original location of the path to l, translating the actions occurring *before* the loop, one transition from l to l, translating the actions occurring *in* the loop, and one transition from l to the path's target, translating the actions occurring *after* the loop. Of course, the two last transitions need to have a guard representing the loop condition.

The inconvenience of such an approach lies in the case of a communication action occurring *after* the loop: When no other automaton would be ready to perform the according co-action, then already the first of the three transitions should not be taken. "Not being ready to synchronize" however, is not implementable as a boolean condition on a transition.

Therefore, the second way is to translate a path containing a **while** action as usual to a single transition, and to translate the loop to an UPPAAL assignment of the form "$V_0 := F_0()$", where V_0 is only a dummy variable, and $F()$ is an UPPAAL function which contains a corresponding **while** loop. An important restriction of this approach is that it could only be applied if the loop does not contain any **wait** actions.

Definition of local variables. In our translator, all variables of a given ATLANTIF module are translated into *global* variables of the resulting network of timed automata. This solution would not be satisfactory if we were to weaken the restriction of unique variable identifiers, as discussed in Remark 4.15.

In the general case however, it is necessary to translate variables as global, because the ATLANTIF semantics provides access to a variable by different units i.e., different timed automata in the translation may access the same variable. Nevertheless, it could make sense to translate those variables that are only accessed in a single unit into a local variable of the TA corresponding to that unit.

Advanced emulation of multiway synchronizations. One inconvenience in our emulation of multiway synchronization is that the resulting chain of UPPAAL synchronizations may start as soon as the first of its synchronizations becomes possible. For instance, we suppose an ATLANTIF module containing a synchronization on G of the units $u1, u2, u3$, which is emulated by two transitions labelled $G\_1\_1!$ and $G\_1\_2!$ in the automaton $\mathcal{A}_{u1}$ (the translation of $u1$), one transition labelled $G\_1\_1?$ in $\mathcal{A}_{u2}$, and one transition labelled $G\_1\_2?$ in $\mathcal{A}_{u3}$. If the transition on $G\_1\_2?$ in $\mathcal{A}_{u3}$ has a guard formula (on clocks and/or on data) which is not satisfied at the beginning of a chain of synchronizations, then a timelock occurs after the first synchronization.

This section sketches an approach that avoids this by strictly extending the translation defined above with additional locations, transitions, channels, and priorities. The basic idea of this approach is to define a "way back" which is automatically taken when a timelock would occur by our "normal" translator.

Suppose an n-ary synchronization on G (more precisely, we speak in the following about the ith synchronization set of G, which has the cardinality $n > 2$). This is translated as defined in Section 6.1.3: There is one timed automaton containing a chain of auxiliary transitions $G\_i\_1!, \ldots, G\_i\_(n-1)!$, beginning in a location s_1 and continuing along auxiliary commited locations $AUX\_G\_i\_2, \ldots, AUX\_G\_i\_n$. What we need is a way back that could start from any auxiliary location. This way back cannot use the given auxiliary locations, because then we would not be able to define the direction that must be taken from such a location on. Thus, we define the auxiliary commited locations $AUX\_G\_i\_2\_back, \ldots, AUX\_G\_i\_(n-1)\_back$.

Then, transitions must be defined as follows, using these new locations for the way back:

- For each $j \in \{3, \ldots, (n-1)\}$, we define one transition from the location $AUX\_G\_i\_j$ to the location $AUX\_G\_i\_(j-1)\_back$, labelled "$Back\_G\_i\_(j-1)!$" and without guards or assignments.

  These transitions are used to initiate the way back.

- For each $j \in \{3, \ldots, (n-2)\}$, we define one transition from $AUX\_G\_i\_j\_back$ to $AUX\_G\_i\_(j-1)\_back$, labelled "$Back\_G\_i\_(j-1)!$" and without guards or assignments.

  These transitions are used to continue along the way back.

- We define one transition from $AUX\_G\_i\_2\_back$ to s_1, labelled "$Back\_G\_i\_1!$" and without guards or assignments.

  This transition concludes the way back.

- We define one transition from $AUX\_G\_i\_2$ to s_1, labelled "$Back\_G\_i\_1!$" and without guards or assignments.

  This transition is used in case of a blocking in the first auxiliary location, when initiating and concluding the way back are done in a single step.

The other $(n-1)$ timed automata participating in the synchronization chain obviously also need to be extended with auxiliary transitions, with the exception of the automaton containing the last synchronization. In each of them, for each transition labelled $G\_i\_j?$ $(j \in 1..(n-2))$ between a location s'_1 and a location s'_2, we define a transition from s'_2 to s'_1, labelled "$Back\_G\_i\_j?$" and without guards or assignments.

As it is clear from the preceding definitions, our extension needs to define $(n-2)$ new auxiliary channels, named $Back\_G\_i\_1, \ldots, Back\_G\_i\_(n-2)$. Furthermore, we must ensure that the way back can only be initiated if the synchronization chain is actually blocked. Therefore, a prioritiy has to be given to the normal continuation of the chain over a transition leading back. UPPAAL provides the possibility of defining priorities on channels. Thus, we define for all $j \in 1..(n-2)$: $G\_i\_(j+1) > Back\_G\_i\_j$.

When we apply these modifications (which do not yet consider clocks and discrete variables) to the Example 6.1 on page 141, we obtain a translation as shown in Fig. 6.5. Furthermore (and not visible in Fig. 6.5), the channel *Cancel_1_2* has priority over the channel *Back_Cancel_1_1*, and *Cancel_1_3* has priority over *Back_Cancel_1_2*.

During the execution of a synchronization chain, clocks are reset and discrete variables are assigned. But if we take the way back before finishing the chain, we obviously want the clocks and variables to have exactly the values they had before starting the chain. Regarding the discrete variables, we can preceed an assignment to a variable V by another assignment "$Backup\_V := V$", where $Backup\_V$ is an auxiliary variable only defined for this aim; then a corresponding transition in the inverse direction would have the inverse

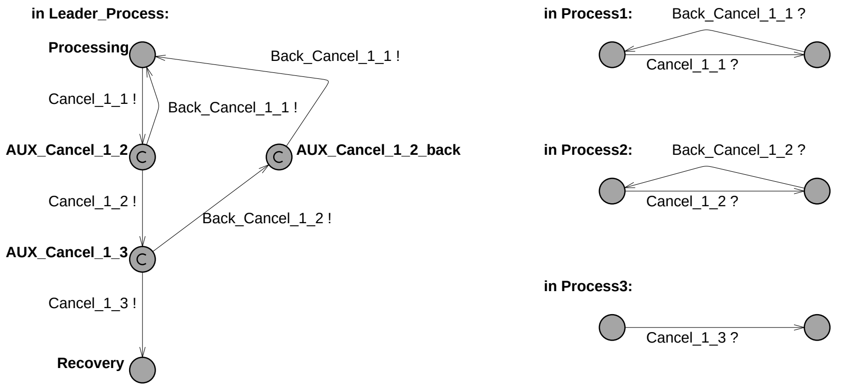

Figure 6.5: Translation fragments in UPPAAL TAs – advanced emulation

assignment "$V := Backup\_V$", giving V back its old value. Note that this is not necessary for the assignments of auxiliary variables that are used in the emulation of offers.

A similar approach cannot be used for clocks, because the UPPAAL TA dialect does not provide the possibility of assigning to a clock the value of another clock (as it is possible e.g., in KRONOS). Instead, assignments that reset clocks to zero have to be made during the *last* synchronization of the chain i.e., after the last possibility to initiate a way back. For this, it is necessary to define all clocks globally (which is possible in UPPAAL).

With these modifications to clock and variable assignments, the final version of the translation from Example 6.1 would be as shown in Fig. 6.6.

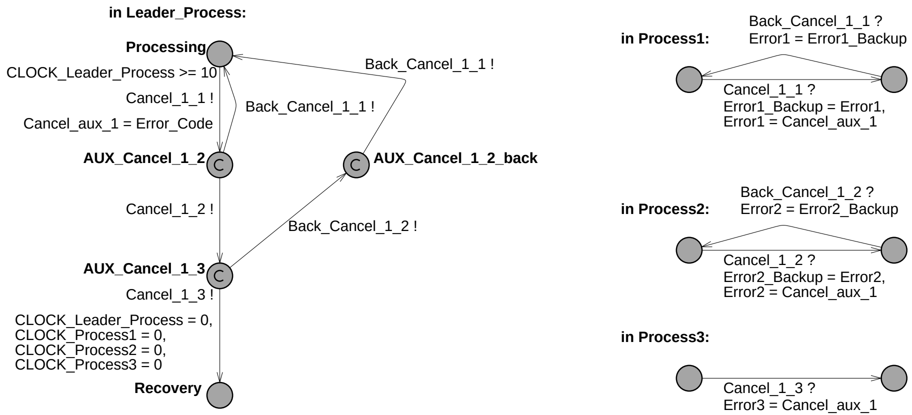

Figure 6.6: Translation fragments in UPPAAL TAs – advanced emulation with clocks and discrete variables

A simpler solution? To solve the problem of synchronization chains that may timelock, instead of the solution we just sketched, one might think that it would be a simpler approach to combine all constraints in one of the transitions for the first synchronization. This is not possible, because synchronization chains are not statically linked to certain transitions, but may take different transitions in each synchronizing TA. Those transitions may of course have different guard formulas, therefore a combined guard formula for one chain can not be determined statically.

Another idea considered but discarded was to define for each possible chain a global boolean variable that takes the value **true** if and only if all guard formulas are satisfied, and then one transition in the first synchronization has a guard depending on this variable. Such a solution is also not practicable, because the satisfaction of guard formulas may change with the elapsing of time. The TA model cannot assign a new value to a discrete variable after the elapsing of time, only during discrete transitions. Therefore, such a variable could not be realized.

6.2 Time Petri nets

6.2.1 Motivation and principles

Choosing TINA-TPN as target model

As described in Section 3.2.3, several tool implementations exist that perform simulation and/or formal verification on the different timed extensions of Petri nets. As in the case of timed automata (cf. Section 6.1.1), each of these tools defines a different dialect for its input language. Among these possible targets, we chose to translate from Atlantif to the TPN dialect of the Tina tool [16] for several reasons:

- Among the extensions of Petri nets, Tina uses *time Petri nets*. As explained on page 30, this approach is the most suitable to express timing aspects.
- The TPN dialect used in Tina has a rich expressive power. It provides several constructs that will be used by the translator, such as
  - inhibitor arcs,
  - priorities on transitions, and
  - data manipulation.
- Tina is a popular, widely known tool, which is an important condition to ensure tool maintenance in the future. Other tools operating on time Petri nets that were created during the last few years are no longer maintained (e.g., Ina [117]).

However, a translator from Atlantif (at least without data) to other TPN dialects is possible in principle and should not be too different from the one to Tina, if the other tools also provide inhibitor arcs and priorities on transitions. Clearly, additional features used in other tools like the *reset arc* of the Roméo TPN dialect [93] could also improve the translator.

The TINA TPN dialect

The definition of a TPN in TINA is a strict extension of Definition 3.9 i.e., a TINA-TPN is a 11-tuple $(P, T, in, out, inh, test, m_0, lab, I_s, Pr, (\mathcal{V}_P, Prec, Ac, \rho_0))$ such that the following is satisfied.

- $(P, T, in, out, m_0, lab, I_s)$ is a TPN as in Definition 3.9.
- *inh* and *test* are mappings from T to subsets of P, indicating the *inhibitor-places* and the *test-places* of a transition.

  Graphically, for each tuple (p, t) such that $p \in inh(t)$, we draw between p and t an arc, whose end pointing to t has a small circle.
- $Pr \subseteq T \times T$ is a strict partial order relation on the transitions[21].

  Graphically, for each $(t_1, t_2) \in Pr$ (i.e., t_1 has priority over t_2), we draw a directed arc with a dotted line from t_1 to t_2.
- The 4-tuple $(\mathcal{V}_P, Prec, Ac, \rho_0)$ extends the TPN into a *Predicate-/Action-TPN*. $\mathcal{V}_P$ is a set of (global) variables. We write $\mathcal{R}_P$ for the set of all possible stores (stores are defined as for ATLANTIF, cf. Section 4.3.2) on $\mathcal{V}_P$.

  Then the mapping $Prec : T \to (\mathcal{R}_P \to \{\mathbf{true}, \mathbf{false}\})$ defines for each transition a predicate on the values of $\mathcal{V}_P$. Intuitively, this establishes a precondition on the variable values that has to be satisfied in order to take this transition.

  The mapping $Ac : T \to (\mathcal{R}_P \to \mathcal{R}_P)$ defines for each transition a *variable update*. Intuitively, each transition is associated with a function that may redefine the variable values depending on the former values.

  ρ_0 is the initial store on $\mathcal{V}_P$.

Note that TINA also provides other TPN-extensions, notably *stopwatches* and weighted inhibitor- and test-arcs. But not all extensions are supported by all tools of the TINA toolbox, and we need not detail features we will not use.

Semantics. A time Petri net with inhibitor arcs, test arcs, priorities, and predicate-/action-transitions basically extends the semantics definition given in Section 3.2.3. Therefore, in the following we will give the entire definition and indicate the extensions with grey background.

The semantics of a net $(P, T, in, out, inh, test, m_0, lab, I_s, Pr, (\mathcal{V}_P, Prec, Ac, \rho_0))$ is defined by a TLTS of the form $(\Sigma, A, \mathbb{R}_{\geq 0}, \to, (m_0, I_0))$, where $\Sigma, \to$ are the smallest sets satisfying:

[21] Note that in textual as well as in graphical definitions of TINA-TPNs, this relation is only partly defined, for convenience reasons: For a given relation Pr', Pr is defined as the smallest partial order containing Pr'.

- The states of Σ are triples of the form (m, I, ρ), where m is a marking (i.e., a multiset in P), I is a partial function from P to $\mathbb{I}$, and ρ is a store on $\mathcal{V}_P$. The domain of I is given by the *enabled* transitions i.e., the transitions r such that the following holds:

  – $in(r) \subseteq m$

  – $inh(r) \cap m = \varnothing$

  – $test(r) \subseteq m$

  – $(Prec(r))(\rho) = \textbf{true}$

- $(m_0, I_0, \rho_0) \in \Sigma$, where I_0 is the restriction of I_s to the enabled transitions.

- *Timed transition*: If $(m, I) \in \Sigma, t \in \mathbb{R}_{\geq 0}$, and for each $r \in dom(I)$, t is in or below $I(r)$, then $(m, shift(I, -t)) \in \Sigma$ and $(m, I) \xrightarrow{t} (m, shift(I, -t))$.

- *Discrete transition*: If

  – $(m, I) \in \Sigma$,

  – $r \in T$ enabled with m and $0 \in I(r)$, and

  – for all $r' \in (T \setminus \{r\})$ enabled with m and $0 \in I(r)$, the predicate $Pr(r',r)$ does not hold,

  then $(m', I') \in \Sigma$ and $(m, I) \xrightarrow{lab(r)} (m', I')$, where $m' = (m \setminus in(r)) \cup out(r)$ and I' is defined by $I'(r') = I(r')$ if $r' \in dom(I) \setminus \{r\}$, otherwise $I'(r') = I_s(r')$.

Note that, by definition of $dom(I)$, the elapsing of time can only be blocked by *enabled* transitions at the end of their firing interval. If, for instance, the predicate (i.e., the data condition) of transition t_1 in the net displayed in Fig. 6.7 is not satisfied for the current store ρ i.e., $(Prec(t_1))(\rho) = \textbf{false}$, then time can elapse beyond two time units, and t_2 might later be fired, which would not be possible in a standard TPN without preconditions.

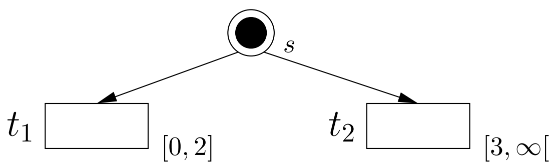

Figure 6.7: Example for a TPN extended with predicate-/action-transitions

In a similar way, priorities apply to a pair of transitions only if both are enabled and within their firing interval.

Implementation of predicate-/action-transitions in TINA. Important for our translator, the TINA tool implements predicate-/action-transitions by C code, in a file separate from the TPN (including the inhibitor and test arcs and the priorities) itself. The 4-tuple $(\mathcal{V}_P, Prec, Ac, \rho_0)$ is represented as follows:

- The set $\mathcal{V}_P$ corresponds to a single variable of a structured type "*key*", with one field for each $V \in \mathcal{V}_P$.
- The mapping *Prec* corresponds to one C function for each $Prec(t)$ $(t \in T)$, with a single input parameter of type *key* and a return value of type *bool*. For those $t \in T$ where $Prec(t)$ is constantly true (i.e., the transition does not have a condition), no C functions need to be defined.
- The mapping *Ac* corresponds to one C function for each $Ac(t)$ $(t \in T)$, with a single input parameter of type *key* and also a return value of type *key*. For those $t \in T$ where $Ac(t)$ is the identity function (i.e., the transition does not manipulate data), no C functions need to be defined.
- The initial store ρ_0 corresponds to one C function without parameters and a return value of type *key*.

Problems to overcome

The following paragraphs list the two most important aspects of the ATLANTIF constructs for which no direct translation to TINA exists. Instead, emulations had to be implemented. Both problems concern aspects related to the ATLANTIF timed structures; the concurrency related constructs did not raise similar difficulties.

Composition of time Petri nets. Necessarily, our translator will compose Petri nets representing units to a Petri net representing a whole module. In the untimed case, the composition of Petri nets consists of carrying over all places and calculating the product of transitions for each label i.e., if the first net contains two transitions r_1 and r_2 with the same label G, and the second net contains one transition r_3 labelled G (as well as transitions with other labels), then the composition contains two $(2 * 1 = 2)$ transitions labelled G: One with in-places $in(r_1) \cup in(r_3)$ and out-places $out(r_1) \cup out(r_3)$, and another one with in-places $in(r_2) \cup in(r_3)$ and out-places $out(r_2) \cup out(r_3)$. Fig. 6.8 shows a small example for such a composition.

A composition of time Petri nets is not a trivial extension of the untimed case, given that firing intervals in T-TPN (cf. Section 3.2.3 on page 30) already represent a concurrent structure. For instance, suppose we want to compose the two transitions shown in Fig. 6.9 *(i)* (where "$|[G]|$" means that the composition is defined by a synchronization on G; following the notation borrowed from LOTOS).

One could intuitively think that Fig. 6.9 *(ii)* is a possible solution for such a composition, where just the intersection of the two intervals $[3, \infty[$ and $[0, 5]$ is calculated for a composed

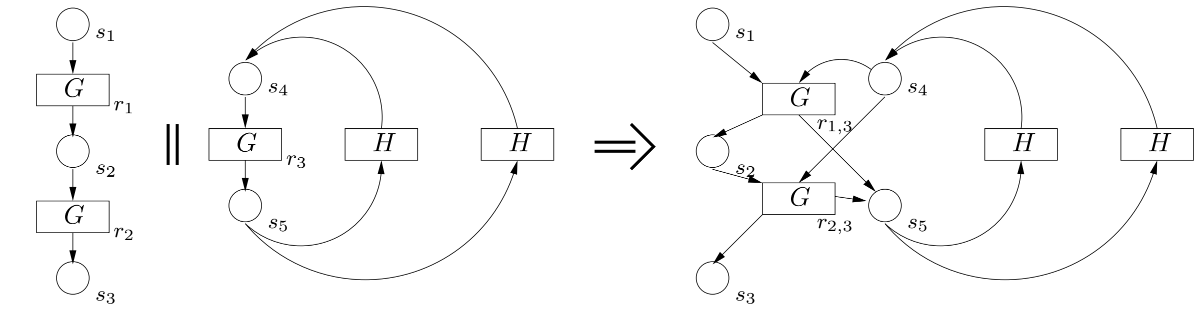

Figure 6.8: Example for the composition of untimed Petri nets

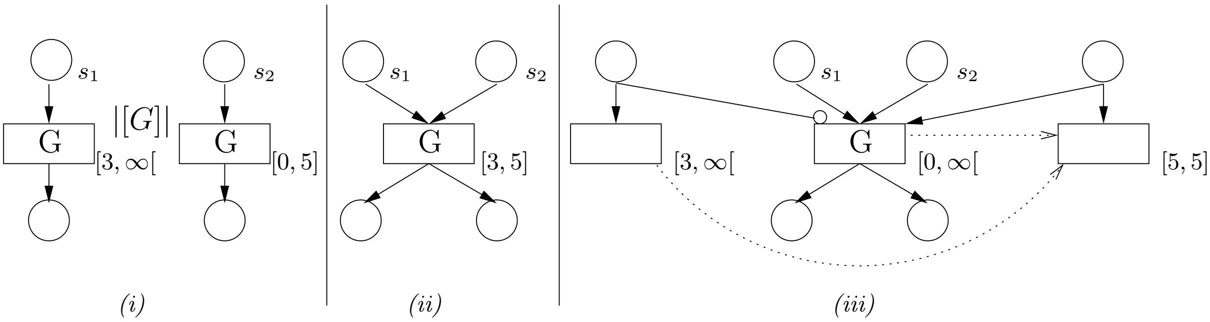

Figure 6.9: Composition of time Petri nets (sketch)

transition. It is easy to see that such a construction would only be correct if tokens in s_1 and in s_2 always arrive at the same time. For instance, if a token arrives in s_2 six time units before a token arrives in s_1, then a synchronization should not be possible. But the solution of *(ii)* would allow a synchronization after three more time units, which is wrong.

Generally speaking, the problem is that firing intervals in T-TPN (cf. Section 3.2.3 on page 30) as they are used in Tina already represent a concurrent structure, which differs from the approach of Atlantif (cf. Section 3.5.1 on page 44). It is therefore necessary to introduce auxiliary unlabelled transitions and auxiliary places that represent time constraints of the unit level.

This intuition has been formalized in [20], where the authors prove in their Theorem 1 that a composition of several TPNs with priorities is possible if each of them satisfies the two following conditions:

1. No labelled transition has a firing interval other than $[0, \infty[$.
2. No priority relation may be defined between two labelled transitions.

Therefore, we have to find a solution for a translation of units that satisfies these conditions; the result is sketched in Fig. 6.9 *(iii)*.

Expressing weak deadlines by strong deadlines. In Tina, the firing intervals have strong deadlines (cf. Section 3.5.1 on page 46), whereas Atlantif uses both strong and weak deadlines. The problem of correctly translating ATLANTIF constructs with weak deadlines is solved at the same time as the problem of composition, where constructions using priorities and inhibitor arcs on the additional unlabelled transitions allow to express weak as well as strong deadlines.

Intuition of the translation approach

An ATLANTIF module is translated into a single TPN. To this end, each unit is translated into a preliminary TPN, then all TPNs are composed according to the synchronizers.

In each unit, 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, one for each execution path and with a label corresponding to the communication action of this execution path if any.

Time constraints are represented by auxiliary places and unlabelled transitions, while the labelled transitions themselves are not constrained in time. The idea for such a separation is taken from [20]. Starting and stopping by synchronizers is translated by modifications in the out-places of the according transitions.

Note that all TPN obtained by the translation of units are 1-bound i.e., none of them can have a place containing more than one token.

6.2.2 Restrictions

Our ATLANTIF to TINA translator only works for a subset of ATLANTIF. The reasons are similar to those described in Section 6.1.2 for UPPAAL. However, the subset of ATLANTIF that can be translated to TINA-TPNs is different from the subset that can be translated to UPPAAL, as the following list shows:

General restrictions

- Only modules with **dense** time domains are translated, because time Petri nets in the TINA dialect use dense time. Translating an untimed ATLANTIF module would however be possible – simply by a "time" Petri net without firing intervals.

- Function and type definitions have only a limited translation: The ATLANTIF code may use external declarations in C code (cf. the syntax definition in Appendix C), in which case these declarations are simply copied. Otherwise, variables are limited to the types **int** and **bool**. However, within the limits of what can be expressed in C, user-defined types and functions of ATLANTIF could also be translated. Very similar translations already exists, for instance the TRAIAN tool for LOTOS NT [116].

Actions

- Non-deterministic assignments are not supported, because, in general, they would induce an infinite number of transitions, which is not feasible. A translation would be possible if the variable type(s) and/or the condition limited the number of possible assigments to be finite.

- Each timed expression E (i.e., E occurring in a time window or as the parameter of a **wait** action) must be an integer constant (although non-integer constants could also be accepted, using the same argument as in Section 6.1.2).

- Time windows may only be intervals. A translation of intersections ("**and**") would be possible without problem, but given the restriction to constant timed expressions, the usage of conjunctions makes no sense (because the intersection of two intervals could be expressed by a single interval in the first place). A translation of unions ("**or**") would be possible, using the decompositions described in Proposition 4.6 on page 97.

- For the same reasons as given in Section 6.1.2, an execution path of a multibranch transition must not contain a condition expression E such that $use(E)$ contains a variable V that occurs as a reception offer "?V" earlier on this execution path.

- **while** loops must not contain **wait** actions. This restriction is due to the fact that timed constraints must be constants, whereas the number of cycles taken in a **while** loop is clearly variable. It could be partially lifted by decomposing execution paths with **while** loops into several transitions.

- In an execution path, there must not be a **case** or an **if** action after a **while** loop, because such a loop could modify variables on which the condition of a **case** or an **if** action depends. In this case, it would not be possible to statically calculate a precondition for the execution path.

- **case** actions are only translated if all patterns are integer constants. Again, a more complete translation would not make theoretical difficulties.

- Offers have to have one of the forms "?V", "?**any** T", "!E". The translation of offers with constructor patterns is probably difficult, but possible when user-defined types of ATLANTIF are translated to TINA. A translation of offers of the form "?(P **where** E)" seems also possible by extending the conditions of translated transitions.

Synchronizers

- For each synchronizer G, all offer lists of communications on G must match in type and cardinality. Releasing this restriction would be possible, but is not yet implemented in the translator.

- For each synchronization set $\mathcal{U} \in \mathit{sync}(G)$ of a synchronizer G with offer cardinality n, for each $i \in 1..n$, there must be exactly one unit $u \in \mathcal{U}$ in which the ith offer of every communication by G is an emission offer.

  This restriction ensures that in each synchronization on an offer, the synchronization value can be calculated deterministically. Otherwise, it would be necessary to generate each possible value, which raises the same problems as a non-deterministic assignment.

- Asynchronous termination cannot be translated i.e., for each synchronizer G, for each $u \in \mathit{stop}(G)$, there must be $u \in \mathcal{U}$ for each $\mathcal{U} \in \mathit{sync}(G)$.

  This restriction is not strictly necessary, but a translation of asynchronous terminations would require an exponential number of auxiliary transitions.

6.2.3 Definition of the translator

We suppose an Atlantif module M that satisfies the static semantics constraints of Chapter 4 and the restrictions listed in Section 6.2.2. As in Chapter 4, we note $\mathbb{G}$ for the set of synchronizers/gates, $\mathcal{V}$ for the set of variables, and $\mathbb{U} = \{u_1, \ldots, u_m\}$ for the set of units.

Translation of one unit into one time Petri net

This section defines the translation of a unit u to a TPN $\mathcal{R}_u$. We suppose the discrete states of u to be $s_0, \ldots, s_m$. The construction of $\mathcal{R}_u$ is performed in two steps.

Step one: The TPN main structure. We begin by defining the function
$\mathit{trans}_p : \mathcal{A} \times (\mathcal{S}_u \times \mathcal{S}_u \times \mathbb{L}_1 \times \mathbb{I} \times \mathbf{Bool} \times \mathcal{E} \times \mathcal{A}) \to \mathcal{P}((\mathcal{S}_u \times \mathcal{S}_u \times \mathbb{L}_1 \times \mathbb{I} \times \mathbf{Bool} \times \mathcal{E} \times \mathcal{A}))$, which evaluates the execution paths of multibranch transitions similar to the function *trans* defined for timed automata. The sets $\mathcal{A}, \mathcal{E}$ contain all actions and expressions respectively that can be constructed from the Atlantif grammar (cf. Table 4.3). The elements (I, O, l, w, x, E, A) of $(\mathcal{S}_u \times \mathcal{S}_u \times \mathbb{L}_1 \times \mathbb{I} \times \mathbf{Bool} \times \mathcal{E} \times \mathcal{A})$ each represent one execution path of a multibranch transition, from which we will derive one TPN transition: I and O will correspond to the input- and output-places, l to the label, w to the timing constraint, x to the semantic modality (**true** for **must** and **false** for **may**), E to the path's condition, and A to the actions concerning data manipulation. We will also simply write Π for these 7-tuples. trans_p is formally defined by cases on its first argument, as shown in Fig. 6.10.

Note that trans_p is defined only for those actions accepted by our translator (cf. Section 6.2.2). In particular, it does not cover nondeterministic assignments and communication actions with composed time windows.

The auxiliary function *update_exp'* is almost identical to the function *update_exp* in the case of timed automata (cf. Section 6.1.3): Instead of list of assignments, the second

$$
\begin{aligned}
&\mathit{trans}_p(\mathbf{null}, \Pi) \stackrel{\text{def}}{=} \{\Pi\}\\
&\mathit{trans}_p(V_0\,\texttt{,}\,\ldots\,\texttt{,}V_n\ \texttt{:=}\ E_0\,\texttt{,}\,\ldots\,\texttt{,}E_n, (I, O, l, w, x, E, A)) \stackrel{\text{def}}{=}\\
&\qquad\qquad\qquad\qquad \{(I, O, l, w, x, E, A\,\texttt{;}\,V_0\,\texttt{,}\,\ldots\,\texttt{,}V_n\ \texttt{:=}\ E_0\,\texttt{,}\,\ldots\,\texttt{,}E_n)\}\\
&\mathit{trans}_p(\mathbf{reset}\ V_0\,\texttt{,}\,\ldots\,\texttt{,}V_n, \Pi) \stackrel{\text{def}}{=} \{\Pi\}\\
&\mathit{trans}_p(\mathbf{wait}\ n, (I, O, l, w, x, E, A)) \stackrel{\text{def}}{=} \{(I, O, l, \mathit{shift}(w, n), x, E, A)\}\\
&\mathit{trans}_p(G\ O_1 \ldots O_n\ \mathbf{must\ in}\ W, (I, O, l, [m, m], x, E, A)) \stackrel{\text{def}}{=}\\
&\qquad\qquad\qquad\qquad \{(I, O, G, \mathit{shift}(W, m), \mathbf{true}, E, A\,\texttt{;}\,G\ O_1 \ldots O_n)\}\\
&\mathit{trans}_p(G\ O_1 \ldots O_n\ \mathbf{may\ in}\ W, (I, O, l, [m, m], x, E, A)) \stackrel{\text{def}}{=}\\
&\qquad\qquad\qquad\qquad \{(I, O, G, \mathit{shift}(W, m), \mathbf{false}, E, A\,\texttt{;}\,G\ O_1 \ldots O_n)\}\\
&\mathit{trans}_p(\mathbf{to}\ s', (I, O, l, w, x, E, A)) \stackrel{\text{def}}{=} \{(I, \{s'\}, l, w, x, E, A)\}\\
&\mathit{trans}_p(\mathbf{stop}, (I, O, l, w, x, E, A)) \stackrel{\text{def}}{=} \begin{cases} \{(I, \varnothing, l, w, x, E, A)\} & \text{if } l \neq \varepsilon \\ \varnothing & \text{otherwise} \end{cases}\\
&\mathit{trans}_p(A_1\,\texttt{;}\,A_2, \Pi) \stackrel{\text{def}}{=} \{\mathit{trans}_p(A_2, (I, O, l, w, x, E, A)) \mid (I, O, l, w, x, E, A) \in \mathit{trans}_p(A_1, \Pi), O = \varnothing\} \cup\\
&\qquad\qquad\qquad\qquad \{(I, O, l, w, x, E, A) \in \mathit{trans}_p(A_1, \Pi) \mid O \neq \varnothing\}\\
&\mathit{trans}_p(\mathbf{select}\ A_0\,\texttt{[]}\,\ldots\,\texttt{[]}\,A_n\ \mathbf{end}, \Pi) \stackrel{\text{def}}{=} \textstyle\bigcup_{i \in 0..n} \mathit{trans}_p(A_i, \Pi)\\
&\mathit{trans}_p(\mathbf{case}\ E_0\ \mathbf{is}\ P_0\texttt{->}A_0\,\texttt{|}\,\ldots\,\texttt{|}\,P_n\texttt{->}A_n\ \mathbf{end}, (I, O, l, w, x, E, A)) \stackrel{\text{def}}{=}\\
&\qquad\qquad \textstyle\bigcup_{i \in 0..n} \mathit{trans}_p(A_i, (I, O, l, w, x, \tilde{E}, A))\\
&\qquad (\text{where } \tilde{E} = E \wedge (\mathit{update\_exp}'(E_0, A) = P_i) \textstyle\bigwedge_{j \in 0..(i-1)} \neg \mathit{update\_exp}'(E_0, A) = P_j)\\
&\mathit{trans}_p(\mathbf{if}\ E_0\ \mathbf{then}\ A_1\ \mathbf{else}\ A_2\ \mathbf{end}, (I, O, l, w, x, E, A))) \stackrel{\text{def}}{=}\\
&\qquad\qquad \mathit{trans}_p(A_1, (I, O, l, w, x, (E \wedge \mathit{update\_exp}'(E_0, A)), A)) \cup\\
&\qquad\qquad \mathit{trans}_p(A_2, (I, O, l, w, x, (E \wedge \neg \mathit{update\_exp}'(E_0, A)), A))\\
&\mathit{trans}_p(\mathbf{while}\ E_0\ \mathbf{do}\ A\ \mathbf{end}, (I, O, l, w, x, E, A)) \stackrel{\text{def}}{=} \{(I, O, l, w, x, E, A\,\texttt{;}\,\mathbf{while}\ E_0\ \mathbf{do}\ A\ \mathbf{end})\}
\end{aligned}
$$

Figure 6.10: The mapping trans_p

argument is a sequential composition of assignments, communication actions, and loops. Because of the restrictions listed in Section 6.2.2, the latter two can be ignored. Therefore, the update of an expression can be calculated as above in *update_exp*.

With the function $trans_p$, it is possible to define the set

$$\mathcal{T}_u \stackrel{\text{def}}{=} \bigcup_{i\in 0..m} trans_p(act(s_i), (\{s_i\}, \varnothing, \varepsilon, [0,0], \mathbf{false}, \mathbf{true}, \mathbf{null})),$$

which corresponds to all possible execution paths in unit u. Then, the time Petri net $\mathcal{R}'_u = (P'_u, T'_u, in'_u, out'_u, m_0{}'_u, lab'_u, I_s{}'_u)$ is defined as follows (cf. Definition 3.9):

- The set of places P'_u contains one element for each discrete state among $s_0, \ldots, s_m$, using the same identifier.
- The set of transitions T'_u contains one element t_Π for each element $\Pi \in \mathcal{T}_u$.
- The mappings in'_u, out'_u, and lab'_u are defined by $in'_u(t_\Pi) \stackrel{\text{def}}{=} I$, $out'_u(t_\Pi) \stackrel{\text{def}}{=} O$, and $lab'_u(t_\Pi) \stackrel{\text{def}}{=} l$ respectively (with $\Pi = (I, O, l, w, x, E, A)$).

  The value of $I_s{}'_u(t_\Pi)$ depends on the value of l: If $l \neq \varepsilon$, then $I_s{}'_u(t_\Pi) \stackrel{\text{def}}{=} [0, \infty[$, otherwise $I_s{}'_u(t_\Pi) \stackrel{\text{def}}{=} w$.
- If u is among the initial units of M, then the multi-set $m_0{}'_u$ equals $\{s_0\}$; otherwise it is empty.

Clearly, in the general case, the TPN $\mathcal{R}'_u$ does not correctly represent the timed behaviour of the unit u', because it always uses strong deadlines (cf. Section 3.5.1), even if u' does not. For this reason, we also continue to refer to the set $\mathcal{T}_u$, which contains information about u that is not yet included in the TPN.

Step two: The TPN auxiliary structures. The TPN $\mathcal{R}_u$ is obtained by extending $\mathcal{R}'_u$ with additional places and transitions with respect to two objectives:

- Emulation of weak deadlines
- *Externalizing* firing intervals i.e., expressing time constraints with auxiliary transitions and thus enabling the labelled transitions to be defined with intervals $[0, \infty[$.

As we indicated above (page 151), the second objective will be essential for the composition. The construction of auxiliary places and transitions is done for each element $\Pi = (I, O, l, w, x, E, A) \in \mathcal{T}_u$ such that $l \neq \varepsilon$[22]. Depending on w (the timing constraint, which has the form of an interval) and x (the semantic modality), six basic cases have to be distinguished (supposing $I = \{s\}$ with $s \in P'_u$):

[22]Those elements with $l = \varepsilon$ do not induce auxiliary transitions. By construction, $w = [m, m]$ with $m \in \mathbb{N}$, and by definition of $\mathcal{R}'_u$, $I_s{}'_u(t_\Pi) = w$

1. $w = [0, \infty[$, x arbitrary:

   Thus, no time restriction applies, and therefore, no auxiliary constructs have to be created.

2. $w = [n', \infty[$, $n' > 0$, x arbitrary:

   This case applies, for instance, if the corresponding execution path contains an action "**wait** n" before a communication with a time window "`[`m`, ... [`" (where $n' = n + m$). In order to externalize this constraint, we define a new place s_1, which always gets a token at the same time as s gets a token (i.e., each transition with s as an out-place gets also s_1 as an out-place), and a new unlabelled transition t with the firing interval $[n', \infty[$, s_1 as in-place and no out-places. Moreover, transition t_Π receives s_1 as inhibitor-place (cf. Section 6.2.1). Schematically, this case is shown in Fig. 6.11.

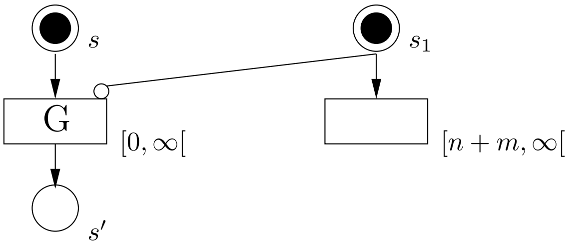

Figure 6.11: Case 2: schema for "**wait** n; G **may in** `[`m`, ... [`" (with $n + m > 0$)

   It can be seen that the inhibitor arc prevents transition t_Π from firing for at least n' time units.

3. $w =]n', \infty[$, x arbitrary:

   This case applies, for instance, if the corresponding execution path contains an action "**wait** n" before a communication with a time window "`]`m`, ... [`" (where $n' = n + m$). In order to externalize this constraint, similarly to the case before, we define a new place s_1, which always gets a token at the same time as s gets a token, and a new unlabelled transition t with the firing interval $]n', \infty[$, s_1 as in-place and no out-places. Moreover, transition t_Π receives s_1 as inhibitor-place. Schematically, this case is shown in Fig. 6.12.

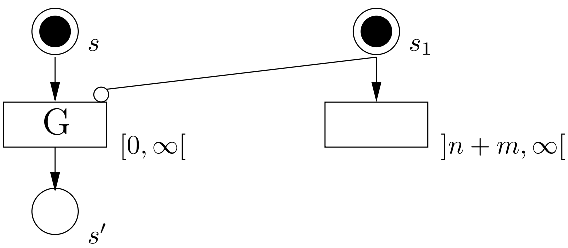

Figure 6.12: Case 3: schema for "**wait** n; G **may in** `]`m`, ... [`" (with $n + m \geq 0$)

4. $w = [0, k]$, $x =$ **false** (**may** modality):

   This case applies, for instance, if the corresponding execution path contains no **wait** action and a communication with a time constraint "**may in** `[0,`k`]`". In order to externalize this constraint, we define a new place s_1, which always gets a token at the same time as s gets a token, and a new unlabelled transition t with the firing interval $]k, \infty[$, s_1 as in-place and no out-places. Moreover, transition t_Π receives s_1 as in-place and a priority (cf. Section 6.2.1 on page 149) lower than t. Schematically, this case is shown in Fig. 6.13.

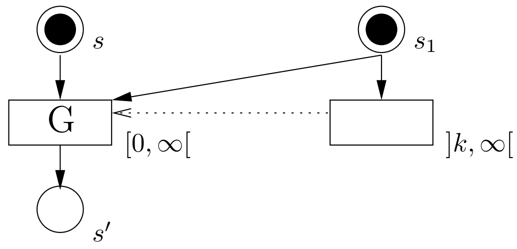

Figure 6.13: Case 4: schema for "G **may in** `[0,`k`]`" (with $k \geq 0$)

   It can be seen that transition t_Π can only be fired during k time units; afterwards the priority hinders t_Π from being fired until t is fired, and then t_Π cannot be fired because there is no token in s_1. Note that time elapsing is never blocked in this construction, thus correctly emulating the weak deadline.

5. $w = [0, k[$, $k > 0$, $x =$ **false** (**may** modality):

   This case applies, for instance, if the corresponding execution path contains no **wait** action and a communication with a time constraint "**may in** `[0,`k`[`". In order to externalize this constraint, similarly to the case before, we define a new place s_1, which always gets a token at the same time as s gets a token, and a new unlabelled transition t with the firing interval $[k, \infty[$, s_1 as in-place and no out-places. Moreover, transition t_Π receives s_1 as in-place and a priority lower than t. Schematically, this case is shown in Fig. 6.14.

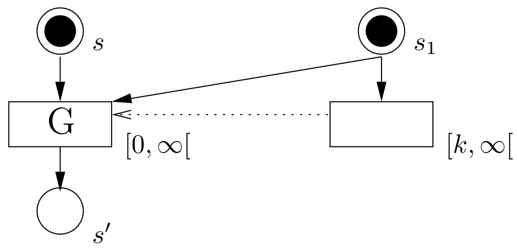

Figure 6.14: Case 5: schema for "G **may in** `[0,`k`[`" (with $k > 0$)

6. $w = [0, k]$, $x =$ **true** (**must** modality):

   This case applies, for instance, if the corresponding execution path contains no **wait** action and a communication with a time constraint "**must in** `[0,`k`]`". In order to externalize this constraint, we define a new place s_1, which always gets a token at the same time as s gets a token, and a new unlabelled transition t with the firing interval $[k, k]$, s_1 as in-place and no out-places. Moreover, t receives a priority not only lower than t_Π, but lower than to all other auxiliary transitions of t_Π. Schematically, this case is shown in Fig. 6.15.

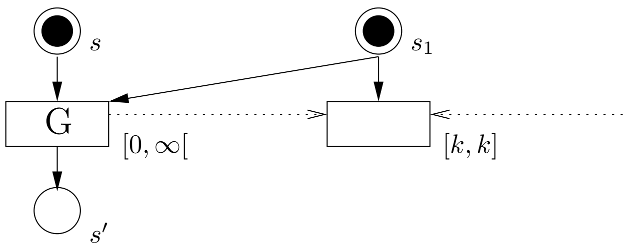

Figure 6.15: Case 6: schema for "G **must in** `[0,`k`]`" (with $k \geq 0$)

It can be seen that as soon as k time unit have elapsed, t has reached its limit and blocks the elapsing of time. If t_Π is enabled, the priorities hinder however t from being fired, which is intended, because firing t would make it impossible to fire t_Π. But this would be semantically incorrect, because time did not elapse *beyond* the time window of the communication, thus firing t_Π still has to be possible.

If, on the other hand, t_Π is not enabled, then t can be fired, and time can continue to elapse.

In [20], priorities are used to represent weak deadlines by time Petri nets. The solutions given in the above list are partially inspired from this approach.

In an excution path with a communication action, either one of the basic cases or a combination among them applies. The following such combinations are possible:

- Case 2 combined with case 4: $w = [n + m, n + k]$ (where $n + m > 0$, $k \geq 0$), $x =$ **false**:

  This case applies, for instance, if the corresponding execution path contains an action "**wait** n" before a communication with a time constraint "**may in** `[`m`,`k`]`". Then we create two auxiliary places s_1, s_2, which always get a token at the same time as s gets a token, one new unlabelled transition t with the firing interval $[n+m, \infty[$, s_1 as in-place, and no out-places, and another new unlabelled transition t' with the firing interval $]n + k, \infty[$, s_2 as in-place, and no out-places. Moreover, t_Π receives s_1 as inhibitor-place, s_2 as in-place, and a priority lower than t'. Schematically, this case is shown in Fig. 6.16.

- Case 3 combined with case 4: $w =]n+m, n+k]$ (where $n+m \geq 0$, $k \geq 0$), $x =$ **false**:

  This case applies, for instance, if the corresponding execution path contains an action "**wait** n" before a communication with a time constraint "**may in** `]`m`,`k`]`". Then we create two auxiliary places s_1, s_2, which always get a token at the same time

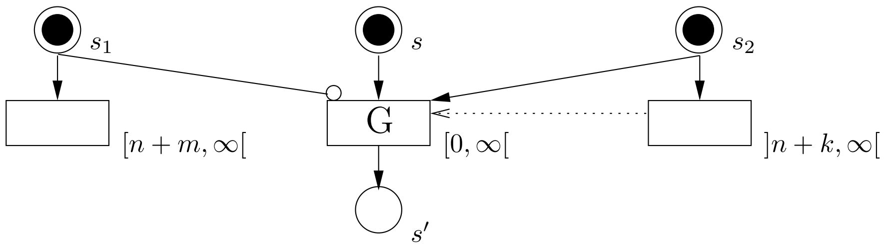

Figure 6.16: Schema for "**wait** n; G **may in** [m,k]" (with $n+m>0, k\geq 0$)

as s gets a token, one new unlabelled transition t with the firing interval $]n+m,\infty[$, s_1 as in-place, and no out-places, and another new unlabelled transition t' with the firing interval $]n+k,\infty[$, s_2 as in-place, and no out-places. Moreover, t_Π receives s_1 as inhibitor-place, s_2 as in-place, and a priority lower than to t'.

- Case 2 combined with case 5: $w = [n+m, n+k[$ (where $n+m>0$, $k>0$), $x =$ **false**:

  This case applies, for instance, if the corresponding execution path contains an action "**wait** n" before a communication with a time constraint "**may in** [m,k[". Then we create two auxiliary places s_1, s_2, which always get a token at the same time as s gets a token, one new unlabelled transition t with the firing interval $[n+m,\infty[$, s_1 as in-place, and no out-places, and another new unlabelled transition t' with the firing interval $[n+k,\infty[$, s_2 as in-place, and no out-places. Moreover, t_Π receives s_1 as inhibitor-place, s_2 as in-place, and a priority lower than to t'.

- Case 3 combined with case 5: $w =]n+m, n+k[$ (where $n+m\geq 0$, $k>0$), $x =$ **false**:

  This case applies, for instance, if the corresponding execution path contains an action "**wait** n" before a communication with a time constraint "**may in**]m,k[". Then we create two auxiliary places s_1, s_2, which always get a token at the same time as s gets a token, one new unlabelled transition t with the firing interval $]n+m,\infty[$, s_1 as in-place, and no out-places, and another new unlabelled transition t' with the firing interval $[n+k,\infty[$, s_2 as in-place, and no out-places. Moreover, t_Π receives s_1 as inhibitor-place, s_2 as in-place, and a priority lower than to t'. Schematically, this case is shown in Fig. 6.17.

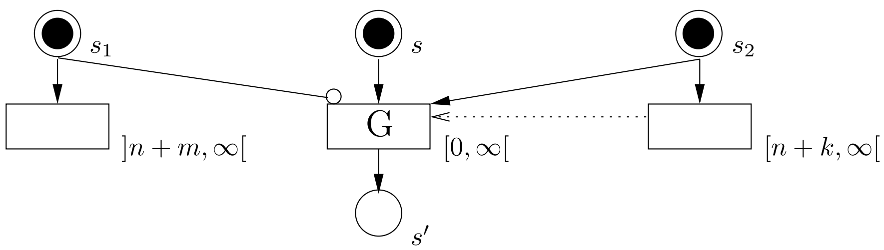

Figure 6.17: Schema for "**wait** n; G **may in**]m,k[" (with $n+m\geq 0, k>0$)

- Case 2 combined with case 6: $w = [n+m, n+k]$ (where $n+m>0$, $k\geq 0$), $x =$ **true**:

This case applies, for instance, if the corresponding execution path contains an action "**wait** n" before a communication with a time constraint "**must in** [m,k]". Then we create two auxiliary places s_1, s_2, which always get a token at the same time as s gets a token, one new unlabelled transition t with the firing interval $[n+m, \infty[$, s_1 as in-place, and no out-places, and another new unlabelled transition t' with the firing interval $[n+k, n+k]$, s_2 as in-place, and no out-places. Moreover, t_Π receives s_1 as inhibitor-place, s_2 as in-place, and t' a priority lower than to t_Π *and* to t. Schematically, this case is shown in Fig. 6.18.

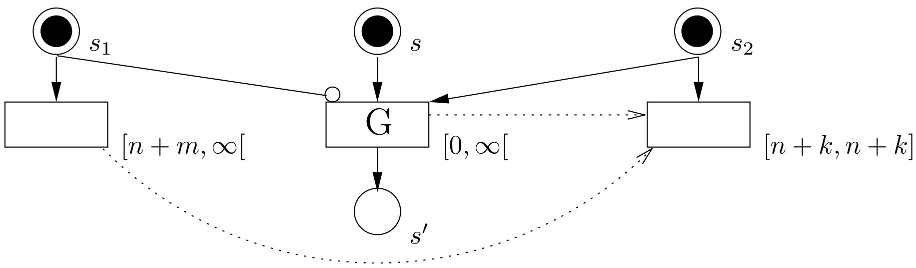

Figure 6.18: Schema for "**wait** n; G **must in** [m,k]" (with $n+m > 0, k \geq 0$)

- Case 3 combined with case 6: $w =]n+m, n+k]$ (where $n+m \geq 0$, $k \geq 0$), $x =$ **true**:

  This case applies, for instance, if the corresponding execution path contains an action "**wait** n" before a communication with a time constraint "**must in**]m,k]". Then we create two auxiliary places s_1, s_2, which always get a token at the same time as s gets a token, one new unlabelled transition t with the firing interval $]n+m, \infty[$, s_1 as in-place, and no out-places, and another new unlabelled transition t' with the firing interval $[n+k, n+k]$, s_2 as in-place, and no out-places. Moreover, t_Π receives s_1 as inhibitor-place, s_2 as in-place, and t' a priority lower than to t_Π *and* to t.

Optimisation. Clearly, the multibranch transition of a discrete state s can induce up to $2 * n$ auxiliary places and transitions, with n being the number of execution paths. Therefore, it makes sense to implement a simple optimisation: If two paths both induce an auxiliary transition with the same time constraint, then they can share a single auxiliary transition and its in-place. This is indeed a realistic case e.g., it applies for the state *Low* of unit *Lamp* in Example 4.1 on page 55. Schematically, this case is shown in Fig. 6.19 (representing the action "**select** G_1 **may in** [0,m[[] G_2 **may in** [m,...[**end select**" without optimisation) and in Fig. 6.20 (representing the same action with optimisation).

From the beginning, the transition labelled "G_1" can be fired, but after m time units, this becomes impossible, because of the priority for the auxiliary transition now firable. Firing the auxiliary transition enables the transition labelled "G_2", thus the behaviour is represented correctly.

Emptying. The definition of auxiliary transitions is still not completely correct, as the following example shows:

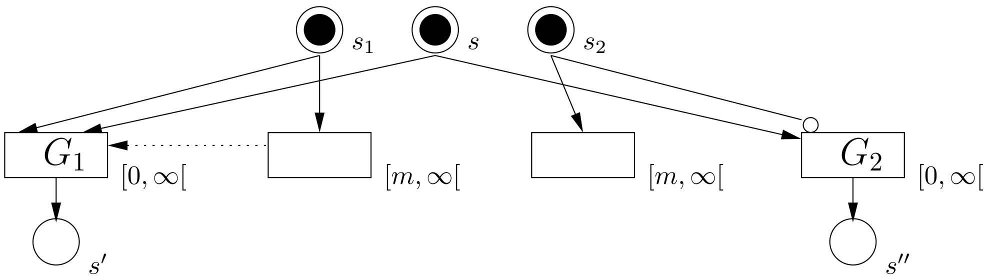

Figure 6.19: Translation for branching action (not optimized)

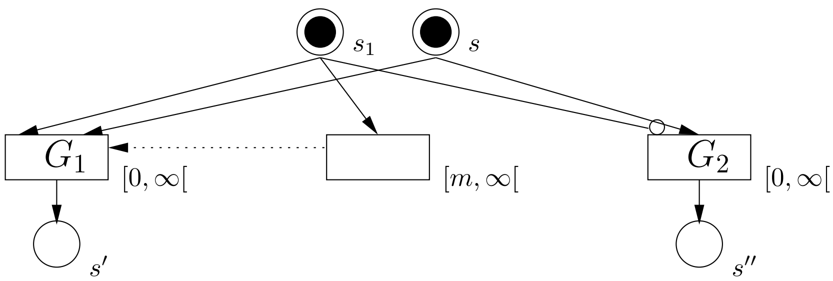

Figure 6.20: Optimization for the translation of Fig. 6.19

Example 6.2. *Consider an* ATLANTIF *unit modelling a sender that consecutively sends messages with four to less than ten time units elapsing between two emissions. If a message is not sent, an error is signalled:*

unit *Irregular_Sender* **is**
  from *Ready*
    select *Send* **in** `[4,10[`; **to** *Ready*
        `[]` *Error* **must in** `[10,10]`; **stop end select**
end unit

According to the preceding schemas, this unit translates to a TPN as shown in Fig. 6.21.

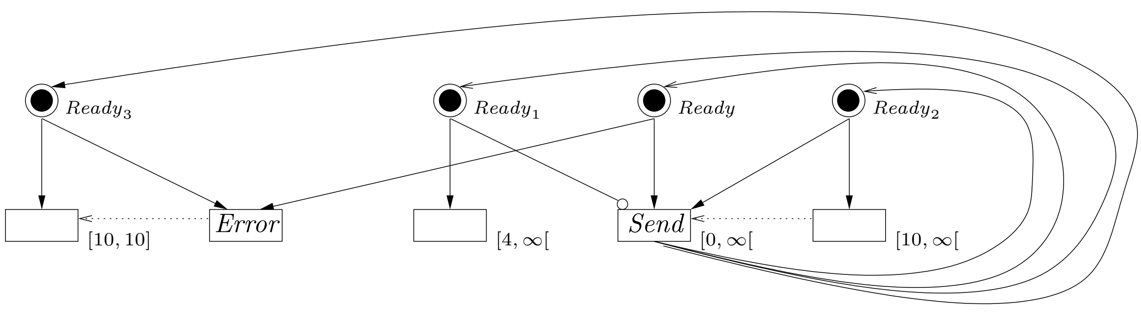

Figure 6.21: Translation for unit *Irregular_Sender* (wrong)

We suppose the following finite run: After an elapsing of five time units, the transition below the place $Ready_1$ *is fired (consuming the token in* $Ready_1$*). Directly afterwards* *(without time elapsing), the transition labelled "Send" is fired. This consumes the tokens in Ready and in $Ready_2$, and creates one new token each in Ready, $Ready_1$, $Ready_2$, and $Ready_3$. Thus, $Ready_3$ then contains two tokens, which is not intended.*

The example shows that it is necessary to empty all auxiliary places that might still have a token after firing a transition corresponding to an execution path. Implementing this is very straightforward, but necessitates the introduction of several more auxiliary places, transitions, and priorities. We describe this with our example and a more general intuition.

In the example, after the transition labelled "*Send*", we add a new out-place. Then we must distinguish between the cases where the token in $Ready_3$ has or has not been consumed. For both cases, we define a new transition that consumes the token in the new out-place and either one token from $Ready_3$ or not, and we give priority to the consumption. Both new transitions have the out-places that belonged before to the transition labelled "*Send*". This correction is shown in Fig. 6.22.

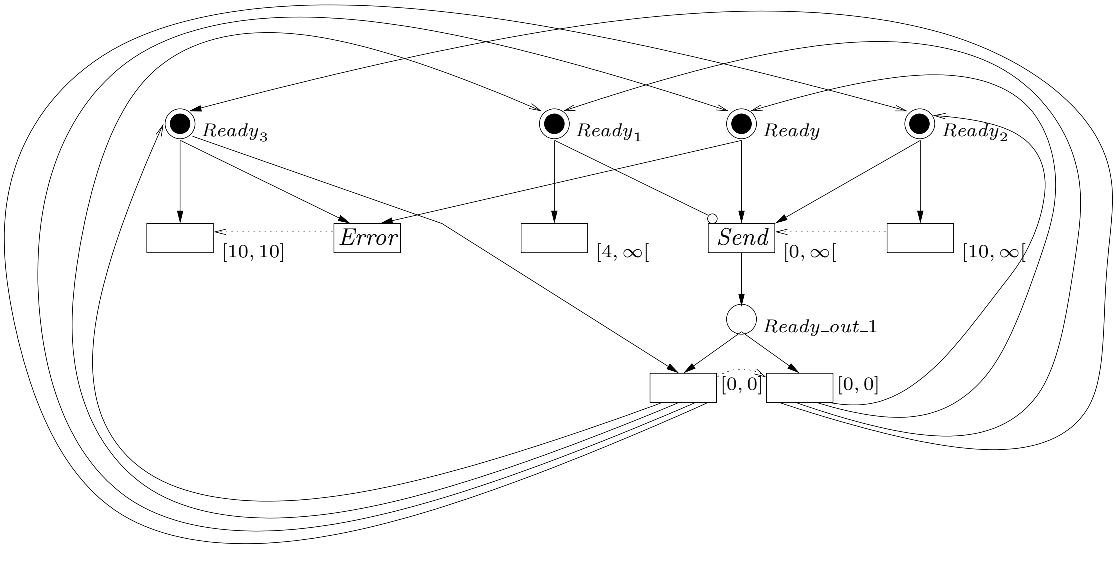

Figure 6.22: Translation for unit *Irregular_Sender* (corrected)

More generally, if a discrete state s allows n execution paths, represented by $\Pi_1, \ldots, \Pi_n \in \mathcal{T}_u$, then for each $i \in 1..n$, the following has to be done:

- One new place *s_out_i* is created, which becomes the only out-place of the transition t_{Π_i} (instead of a set of out-places O_i as defined before).
- Let X be the set of auxiliary places created for the time constraints of the paths $\Pi_1, \ldots, \Pi_{i-1}$ and $\Pi_{i+1}, \ldots, \Pi_n$ (except those also used for Π_i, following the above optimisation). Then for each $Y \subseteq X$, one new unlabelled transition is created, with in-places $Y \cup$ *s_out_i*, out-places O_i, and firing interval $[0, 0]$. Moreover, if $Y \subset Y' \subseteq X$, then the new transition created for Y' has priority over the transition created for Y (which always ensures that the transition that removes all tokens is fired).

A more formal definition of the second step, including the optimisation and the emptying, is provided in the Appendix A.2.1.

Expressions. Expressions can occur either as timed expressions or in conditions, assignments, and offers. By the restrictions (cf. Section 6.2.2), timed expressions are always integer constants. The other kinds of expressions all concern aspects of data manipulation, which, as already indicated, will be represented by C code.

Therefore, as in the translator to UPPAAL (cf. Section 6.1.3), the translation of expressions only concerns trivial syntactical aspects and is therefore not detailed.

Translation of one module into one time Petri net

The TPN $\mathcal{R}_M = (P, T, \mathit{in}, \mathit{out}, \mathit{inh}, \mathit{test}, m_0, \mathit{lab}, I_s, \mathit{Pr}, (\mathcal{V}_P, \mathit{Prec}, \mathit{Ac}, \rho_0))$ is defined as follows:

- The set of places P is the union of all sets of places of $\mathcal{R}_{u_1}, \ldots, \mathcal{R}_{u_m}$, where the TPNs $\mathcal{R}_{u_1}, \ldots, \mathcal{R}_{u_m}$ are translations of the units $u_1, \ldots, u_m$ as defined above in this section (from page 155 on). Without loss of generality, the place and transition identifiers are assumed to be globally unique[23].
- The set of transitions T is defined as the union of the two following sets:
  - T_1, containing all unlabelled transitions from $\mathcal{R}_{u_1}, \ldots, \mathcal{R}_{u_m}$. These correspond to multibranch transition execution paths without communication, and to auxiliary transitions of the unit translation.
  - T_2, the composition of all labelled transitions from $\mathcal{R}_{u_1}, \ldots, \mathcal{R}_{u_m}$, following an algorithm using three concentric loops, shown in Fig. 6.23. Central element of this algorithm are the mappings written C each of which corresponds to one transition in the composition, like the transitions $r_{1,3}, r_{2,3}$ in the example of Fig. 6.8.
- The functions $\mathit{in}, \mathit{out}, \mathit{inh}, \mathit{lab}, I_s, \mathit{Prec}, \mathit{Ac}$[24] are defined for each element of T_1 as before in the corresponding TPN among $\mathcal{R}_{u_1}, \ldots, \mathcal{R}_{u_m}$. For the elements of T_2, they are defined as indicated in Fig. 6.23. The function *test* maps constantly to the empty set.
- The set of initial places m_0 is the union of all sets of initial places of $\mathcal{R}_{u_1}, \ldots, \mathcal{R}_{u_m}$.

[23]Note that the TINA-TPN dialect allows two different places to have the same label, as long as their (internal) identifiers are different.

[24]As described above, the mappings *Prec* and *Ac* are defined by C code in the TINA model. In our translator, we therefore will generate C code from boolean formulas and from ATLANTIF assignment actions and their compositions by "`;`" and by **while** loops. This is clearly a trivial task, thus for the sake of readability we will continue to use ATLANTIF syntax in the remainder of Section 6.2.3.

- The priority relation Pr extends the union of all these relations of $\mathcal{R}_{u_1}, \ldots, \mathcal{R}_{u_m}$ by the following:
  - Let $T_3 \subseteq T_1$ be the set of those transitions t such that $I_s(t) = [0, 0]$ and $out(t) \neq \varnothing$ (which means that t is an auxiliary transition used for place emptying, cf. page 162). Then each transition from T_3 has priority over each transition not in T_3.
  - For each transition $t \in T_1$ such that $I_s(t) = [k, k]$ (for any $k \in \mathbb{N}$), $out(t) = \varnothing$, and $in(t) = \{p\}$ (for any $p \in P$) such that there is a non-empty set $T' = \{t' \in T_2 \mid p \in in(t')\}$ (which means that t is an auxiliary transition for a **must** time limit related to the transitions in T', cf. page 159), we add priorities as follows: t receives lower priority than any other of those auxiliary transition for the elements of T' that are expressing a delay or a **may** time limit.

  These two extensions will be justified by the Examples 6.3 and 6.4.
- The set $\mathcal{V}$ of variables in M (restricted to integers and booleans) translates directly to a set $\mathcal{V}_P$.
- The initial store ρ_0 is the initial store of M according to the ATLANTIF semantics (cf. Section 4.6.3).

The above definition of the composition extends the priority relation, by the emptying having priority over everything else and by auxiliary transitions for **must** time windows having lower priority than certain auxiliary transitions originating in other units. In the Examples 6.3 and 6.4 we will show why this is necessary.

Refinement: unit stopping and starting

To simplify matters, the algorithm presented in Fig. 6.23 does not consider stopping and starting of units. In the following, we describe how the integration of these features modifies the composition.

We note $initial(u) \subseteq P_u$ for the places of the TPN constructed for unit u that represent the initial marking. It contains the place representing the first discrete state of u and the auxiliary places for the firing intervals for the execution paths of this state. Thus, if u is among the initial units of the module, then $initial(u) = m_{0u}$.

For each transition t obtained by composition on a synchronizer G, we apply the following modifications:

- For each $u \in stop(G)$, eliminate the all elements of P_u from the out-places of t.
- For each $u \in start(G)$, add all elements of $initial(u)$ to the out-places of t (by construction, this is a disjoint union).

[25] A'_i and A''_i are (possibly empty) sequential compositions of assignments and **while** actions.

For each synchronizer G:
  For each set $\mathcal{U} \in \mathit{sync}(G)$:
    For each mapping $C : \mathcal{U} \to \bigcup_{i \in 1..n} T_{u_i}$ such that for each $u \in \mathcal{U}$, $C(u) \in T_u$
      and $C(u)$ is labelled G:
      Create a new transition as follows:
        – The in-places are given by $\bigcup_{u \in \mathcal{U}} \mathit{in}(C(u))$, out- and inhibitor-places
          correspondingly.
          (see page 166 for a refinement related to starting and stopping)
        – The label is G if $\mathit{gate}(G) = \mathbf{visible}$, otherwise the transition is unlabelled.
        – The firing interval is $[0,0]$ if $\mathit{gate}(G) \in \{\mathbf{urgent}, \mathbf{silent}\}$,
          otherwise it is $[0, \infty[$
        – Let $A_1, \ldots, A_l$ be the actions of the elements of $\mathit{image}(C)$.
          By definition of trans_p, they have the form
          $(\forall i \in 1..l)\ A_i = A'_i ; G\ O_i^1 \ldots O_i^k ; A''_i$.[25]
          Then the action of the new transition is given by:
          $A'_1 ; \ldots ; A'_l ; \mathit{Resolve\_Offers}(O_1^1 \ldots O_1^k, \ldots, O_l^1 \ldots O_l^k).\mathit{action} ; A''_1 ; \ldots ; A''_l$
          (function *Resolve_Offers* is defined in Fig. 6.24)
        – Finally, the condition is given by
          $\mathit{Resolve\_Offers}(O_1^1 \ldots O_1^k, \ldots, O_l^1 \ldots O_l^k).\mathit{condition} \wedge \left(\bigwedge_{u \in \mathcal{U}} \mathit{Prec}(C(u))\right)$
    end for
  end for
end for

Figure 6.23: Algorithm for composition of transitions in TPNs from ATLANTIF units

$\mathit{Resolve\_Offers}\ (O_1^1 \ldots O_1^k, \ldots, O_l^1 \ldots O_l^k) \stackrel{\text{def}}{=}$
  $\mathit{action} := \textbf{null}$
  $\mathit{condition} := \textbf{true}$
  for each $i \in 1..l$
    $\mathit{current\_offers} := \{O_i^1, \ldots, O_i^k\}$
    choose one O' from *current_offers* that is an emission of the form "$\texttt{!}E$"
    $\mathit{current\_offers} := \mathit{current\_offers} \setminus \{O'\}$
    for each $O'' \in \mathit{current\_offers}$
      $\mathit{current\_offers} := \mathit{current\_offers} \setminus \{O''\}$
      which form has O''?
        when it is "**?any** T", do nothing
        when it is "$\texttt{?}V$", then $\mathit{action} := \mathit{action}\texttt{;}\ V\ \texttt{:=}\ E$
        when it is "$\texttt{!}E'$", then $\mathit{condition} := \mathit{condition} \wedge (E = E')$
    end for
  end for
  return action
  return condition

Figure 6.24: Pseudocode to resolve offers into assignments and a condition (two return values)

Note that this approach does not cover asynchronous termination (i.e., synchronizers that satisfy $(\exists \mathcal{U} \in \mathit{sync}(G))\ \mathit{stop}(G) \setminus \mathcal{U} \neq \varnothing$), which we therefore excluded in Section 6.2.2. The translation of asynchronous termination is technically possible; it would necessitate elimination of all tokens from the places representing the corresponding unit, similar to the "emptying" defined above (cf. page 162). Clearly, a large number of extra transitions would be needed for this deletion.

6.2.4 Discussion

Usage of priorities

Overview of the priorities used. In this paragraph, we will summarize the usage of priorities by our translator and show that they define indeed a partial order. Our translator uses priorities for three different reasons:

1. They ensure the correct functioning of weak deadlines (effective disabling of a labelled transition with the aid of a priority) and of strong deadlines (time-out only when the synchronization is not possible, which is necessary for time determinism).

2. Emptying is always performed for all places that need to be emptied.

3. Emptying directly follows the corresponding labelled transition.

This usage of priorities is made in two different ways: First, "individual" priorities between single transitions are defined for reasons 1 and 2 above. Second, priorities between different "families" of transitions are defined for reason 3 above. In our translator, we can distinguish three such families:

- $\mathcal{T}_\alpha$: non-auxiliary transitions (labelled transitions, those constructed from execution paths without communication, and those corresponding to hidden or urgent synchronizations)
- $\mathcal{T}_\beta$: auxiliary transitions for time limits and delays
- $\mathcal{T}_\gamma$: auxiliary transitions for emptying

Between thoses families, the transitions of $\mathcal{T}_\gamma$ have priority over those from the two other families i.e., in an informal notation, we could write this $\mathcal{T}_\gamma > (\mathcal{T}_\alpha \cup \mathcal{T}_\beta)$.

We can see that the "individual" priorities are either between elements of $\mathcal{T}_\alpha$ and $\mathcal{T}_\beta$ or within $\mathcal{T}_\gamma$; thus the combination of all priorities (individual and family) cannot render impossible the definition of a transitive closure of the priority relation, which is a partial order.

The two following examples illustrate the modifications how our translator handles priorities when composing translations of units into a single TPN.

Example 6.3. *This example illustrates why "emptying transitions" need to have priority over all other transitions, including those created for other units.*

Supposing a very simple sender/receiver model that is given by the Atlantif *module of Fig. 6.25.*

```
module Sender_and_Receiver is
dense time
sync Get_Rdy is Receiver end sync
sync Reset is Receiver end sync
sync Transmission is Sender and Receiver
  end sync

init Sender, Receiver

unit Sender is
  from Processing
    Transmission must in [0,5];
    to Processing
end unit

unit Receiver is
  from Preparing
    select
        Get_Rdy; to Receiving
      [] Reset in [0,10]; to Preparing
    end select
  from Receiving
    Transmission;
    to Preparing
end unit

end module
```

Figure 6.25: Very simple sender/receiver model

In this specification, the unit Sender synchronizes repeatedly on gate Transmission, where zero to five time units may elapse between two synchronizations. The unit Receiver may *need an arbitrary time to Get_Ready, but it can also Reset during the first ten time units after its initialization. After a synchronization on Get_Ready, the receiver may synchronize on Transmission and then reinitialize.*

By the ATLANTIF *semantics we then can derive a run such as the following:*

$$S_0 \xrightarrow{5} S_1 \xrightarrow{Get\_Rdy} S_2 \xrightarrow{Transmission} S_0$$

For this run, we suppose that $S_0 = ([Sen \mapsto Pro, Rec \mapsto Pre], [Sen \mapsto (0, \mathbf{f}), Rec \mapsto (0, \mathbf{f})], \varnothing)$ *(the initial state), that* $S_1 = ([Sen \mapsto Pro, Rec \mapsto Pre], [Sen \mapsto (5, \mathbf{f}), Rec \mapsto (5, \mathbf{f})], \varnothing)$*, and finally that* $S_2 = ([Sen \mapsto Pro, Rec \mapsto Rvg], [Sen \mapsto (5, \mathbf{f}), Rec \mapsto (0, \mathbf{f})], \varnothing)$.

By our translator, this module is translated into the TINA-*TPN shown in Fig. 6.26. In particular, the translator defines the transition with time interval* $[5, 5]$ *(originally from unit Sender) to have lower priority than the two emptying transitions following the synchronization on Get_Ready (originally from unit Receiver).*

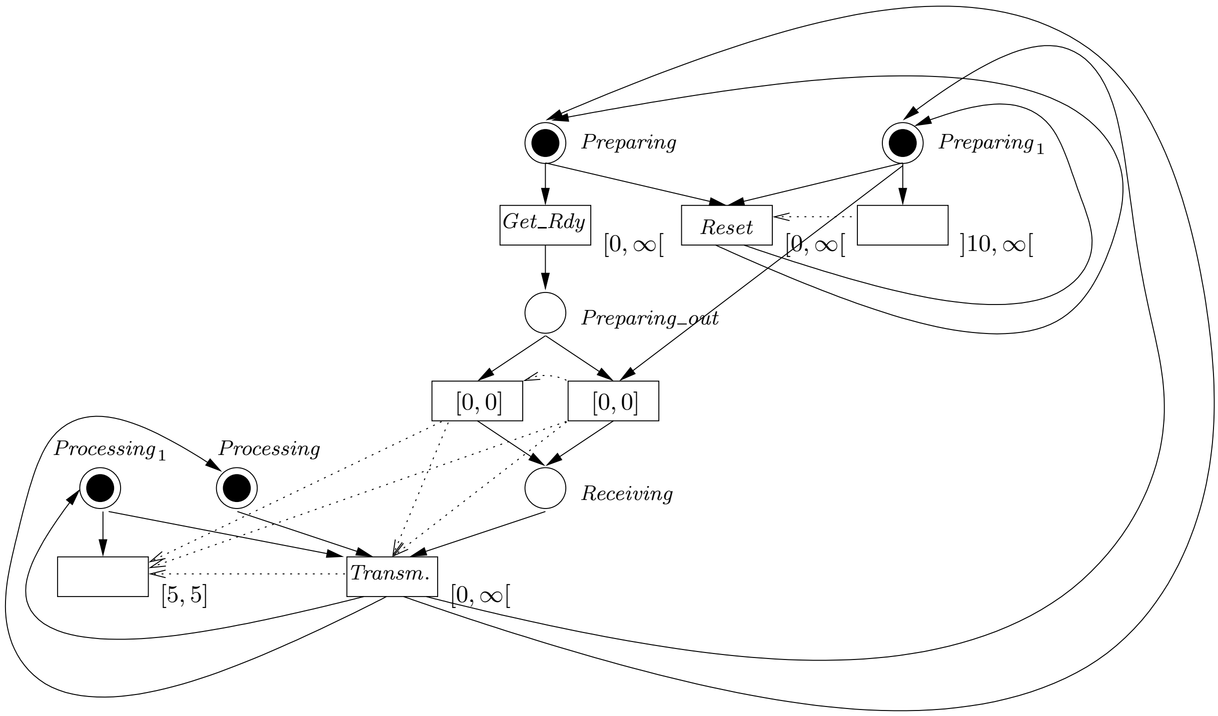

Figure 6.26: Translation of the very simple sender/receiver model

Suppose that after the elapsing of five time units the transition labelled Get_Rdy is fired. Then, if the emptying transitions do not have priority over other transitions, the transition on the lower left (the time-out) could be fired before the emptying transition with the in-places Preparing and Preparing$_1$*. But in this case, a synchronization on Processing would not be possible any more, although only five time units have elapsed.*

Therefore, the priorities on the emptying are needed to ensure a translator in which the semantics are stable.

Example 6.4. *This example illustrates why auxiliary transitions for time windows with modality* **must** *have priority over auxiliary transitions for time windows with modality* **may** *that are created for the same multibranch transition.*

Supposing an even more simple variant of the sender/receiver model from Example 6.3, given by the ATLANTIF *module of Fig. 6.27.*

module *Sender_and_Receiver* **is**
dense time

sync *Transmission* **is**
Sender **and** *Receiver* **end sync**

init *Sender* , *Receiver*

unit *Sender* **is**
from *Processing*
Transmission **must in** [0,5] ; **to** *Processing*
end unit
unit *Receiver* **is**
from *Receiving*
Transmission **may in** [3, . . . [; **to** *Receiving*
end unit
end module

Figure 6.27: Very simple sender/receiver model (variant)

In this specification, the sender is the same as in the preceding example, but the receiver is simplified to repeatedly (spaced by at least three time units) perform synchronizations on the gate Transmission.

By our translator, this module is translated into the TINA-*TPN shown in Fig. 6.28. In particular, the translator defines the transition with time interval* $[5, 5]$ *(originally from unit Sender) to have lower priority than the transitions with time interval* $[3, \infty[$ *(originally from unit Receiver).*

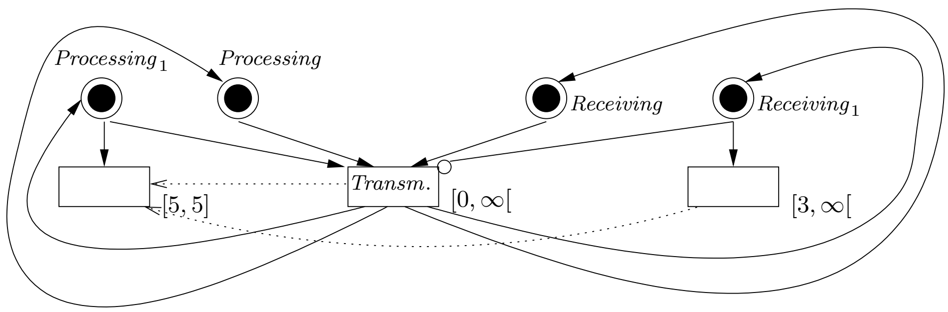

Figure 6.28: Translation of the very simple sender/receiver model (variant)

Suppose that five time units elapse. If there was no priority between the two auxiliary transitions, we could fire the transition with time interval $[5, 5]$*, thus making the firing of the transition labelled Transmission impossible. But this would be wrong, because time did not elapse beyond the strong deadline of five time units.*

Because of the priorities, first the transition with time interval $[3, \infty[$ *must fire, then the transition labelled Transmission, which correctly represents the* ATLANTIF *semantics.*

Impact of the restrictions

Similar to our observation from Section 6.1.4, we can also state here that the list presented in Section 6.2.2, defining the subset of Atlantif that can be translated into time Petri nets is not as restrictive as its length might suggest. Again, it is possible to avoid several of the restrictions by alternative constructs, as already indicated in Sections 6.1.4 and 6.2.2.

Besides such merely technical restrictions, actual limitations are caused by those restrictions applying to offers and to asynchronous terminations. As already in the case of the Uppaal translator, none of the examples in Chapter 5 would be excluded by the restrictions on offers, but three examples use synchronizers with asynchronous termination. In these cases however, an alternative is provided in form of a translation to Uppaal.

Again, the impact of the restrictions is thus limited.

Imprecision problems of the translator

As in the case of the translator to Uppaal, there are problems regarding the preservation of semantics. Beside the same technical issue of ignoring variable resets, the introduction of unlabelled transitions in particular changes the semantics, by adding discrete τ-transitions that are not intended by the semantics rules of Atlantif.

- As in the translator to Uppaal, execution paths (of a multibranch transition) without communication action and synchronizations on silent synchronizers produce unlabelled transitions.
- Moreover, additional unlabelled transitions are introduced by auxiliary transitions expressing time constraints. But unwanted new behaviour (such as a timelock) is avoided in our translator, mainly by the use of priorities (cf. Examples 6.3 and 6.4).
- Finally, additional unlabelled transitions are introduced by the emptying of auxiliary places. With priorities and $[0, 0]$ intervals, it is ensured that those transitions are always fired at once (i.e., before time can elapse and before any other transition is fired) when they can be fired.

Again, these imprecisions do not have an impact on properties expressed in LTL_{-X} if the Atlantif code ensures that taking an ε-transition never resolves a choice.

Comparison of the restrictions and the imprecisions

If we compare the subsets of Atlantif given by the different restrictions that we stated for the Uppaal translator and for the Tina translator, we see that neither of them is a subset of the other.

Two restrictions that we stated for the Uppaal translator are entirely released in the Tina translator, as described in the following:

- Communication actions by synchronizers tagged as **urgent** may have timing constraints.
- Branches in **if** actions may contain **must** time windows.

Several restrictions that we stated for the Uppaal translator are partially released in the Tina translator, as described in the following:

- Within one communication action, offers can be mixed between emissions and receptions, and a synchronization on an offer may contain more than one emission. In both translators however, at least one emission is needed.
- **case** and **while** actions are allowed, although with a limited syntax.
- Functions can be translated, but also with a limited syntax.

Against these advantages of the Tina translator, there is one important shortcoming, which is the absence of a translation for asynchronous termination.

Regarding the semantic imprecisions, the Tina translator does not have the problem of multiple labelled transitions that represent a single synchronization. In particular, unit starting and stopping (by synchronous termination) are translated in a more simple and elegant way than in the Uppaal translator. Nevertheless, both translators introduce auxiliary transitions for different reasons. Clearly, it depends on the Atlantif code to be translated, which translator introduces more of them.

Possible improvements and extensions

The major starting points for improvements to the Tina translators are already given along with some of the restrictions listed in Section 6.2.2 and summarized below:

- Similarly to the corresponding section regarding the Uppaal translator, a more powerful translation of **while** actions could be based on decomposing execution paths containing a loop into three different steps. Such an extension would provide the translation of **while** actions containing **wait** actions.
- The restriction that all occurrences of a given gate must have offers that match in type and cardinality could be lifted by using an approach similar to how Lotos offer synchronizations are processed in the Cæsar tool [59]. Such an extension of the algorithm shown in Fig. 6.23 is mainly a technical question in the implementation, in which this algorithm would only be extended by an additional condition in the head of the third loop.
- More expressive power can be given to patterns that are used in case actions and in reception offers.

6.3 Fiacre

The translator we defined to generate a Fiacre program from an Atlantif module is still a prototype and contains important restrictions. Therefore, we only give an overview in this section, while a more complete definition can be found in the Appendix A.3.

6.3.1 Motivation and principles

A translator to the Fiacre model (cf. Section 3.4.7) is interesting for three main reasons:

- Fiacre has been developed in the context of research projects such as OpenEmbeDD and Topcased, which involve several industrial and academical partners. This large participation indicates that future developement and maintenance of this model will be ensured.
- A software tool called "`flac`" is already available for Fiacre. It translates an untimed Fiacre specification into Lotos code, which can be verified using the Cadp [63] (*Construction and Analysis of Distributed Processes*) toolbox. Thus, verification of an Atlantif specification by Cadp becomes possible.[26]
- When we define a translator from Atlantif to Fiacre, we get a detailed understanding of the differences between these two models. Therefore, a better comparison of their expressive power becomes possible.

Due to fundamentally different approaches of Atlantif and Fiacre regarding real-time syntax and semantics (cf. Sections 3.5.1 and 4.2), the translator we propose in this section and in the Appendix A.3 is limited to *untimed* Atlantif.

6.3.2 Intuition of the translation approach

One Atlantif module M with $\mathbb{U} = \{u_1, \ldots, u_n\}$ translates to one Fiacre program. As Fiacre processes and Atlantif units are very similar structures, we will translate each unit of $\mathbb{U}$ into one process, with each Atlantif multibranch transition being translated into a Fiacre multibranch transition (they share a very similar syntax). The unit hierarchy is not represented in the translation, but instead, we introduce a new hierarchy by regrouping several processes into Fiacre components, which is schematically shown in Fig. 6.29 for an Atlantif module M.

It can be seen from this schema that all those processes representing unit translations are on the lowest hierarchical level. Above, they are regrouped by one component *l2* that contains one synchronization vector which represents all synchronizers. On the topmost

[26] Another tool developed for Fiacre is "`frac`", which translates Fiacre into Tina TPN, but this is less interesting for us, because we have already a direct translator from Atlantif to Tina.

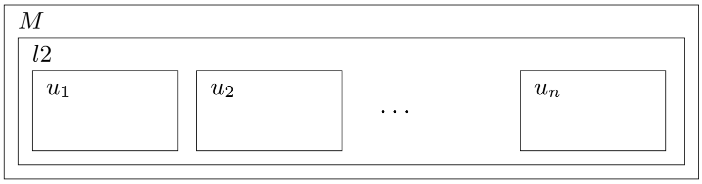

Figure 6.29: Component hierarchy of a generated FIACRE program (schema)

level, one component ensures the renaming of auxiliary ports introduced in *l2*, and ensures the representation of hiding.

Following this structure, the translator is composed of three steps: The construction of the topmost component, followed by an analysis of the synchronizers to determine the emulations that have to be made and the construction of the component *l2* with its synchronization vector, and finally the translations of all units to processes. These steps are detailed in the Appendix A.3.4.

6.4 Tool implementation

In this section, we give a brief overview of a prototype implementation of a tool we created on the basis of the translators defined in this chapter. For convenience, in the following we will write "`atlantif`" if we refer to this tool (whereas we write "ATLANTIF" when referring to the model).

The implementation of `atlantif` uses the method proposed in [62] i.e., we used the SYNTAX [34] system (currently 2,193 lines of code) to encode the ATLANTIF grammar, we used the LOTOS NT [116] language (currently 13,146 lines of code) to define the data types and functions for the translators, and we embedded everything in a C main program (currently 538 lines of code). The architecture of `atlantif` is illustrated in Fig. 6.30, where bold arrows represent the translators provided by our tool.

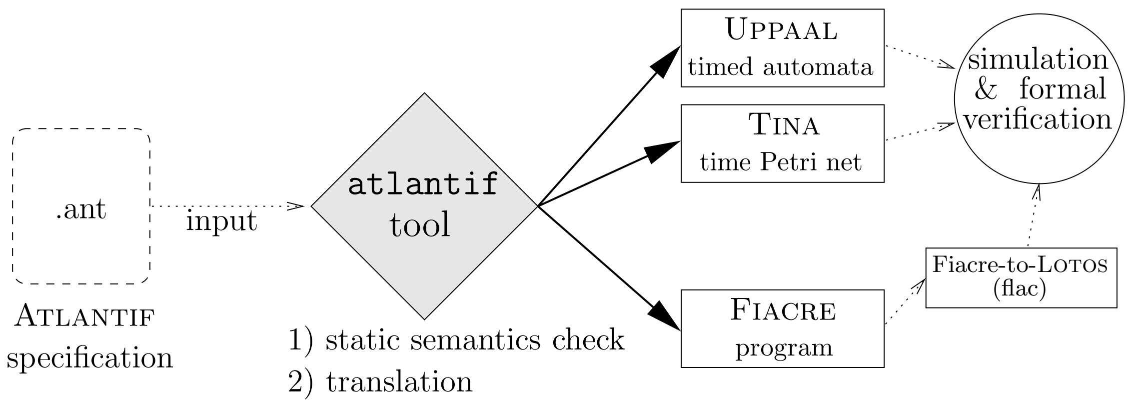

Figure 6.30: Schema for the application of the prototype `atlantif` tool

The input accepted by `atlantif` is one ATLANTIF module as defined in Section 4.2

(Table 4.3). The complete grammar defined with SYNTAX is shown in Appendix C, including a detailed definition of expressions and a preliminary encoding for types and functions.

Given such an input, `atlantif` performs a verification of the static semantic constraints defined in Sections 4.3 to 4.6. Among others, it uses the algorithms given in the Appendix A.1.

If the input satisfies the static semantics of ATLANTIF, one of the three translators defined in this chapter is executed. Error messages are issued if the restrictions for the concerned target language are not met.

We tested the tool on several common benchmark examples in order to further estimate the impact of the restrictions and to compare the sizes of ATLANTIF code with its translations. These examples are the light switch presented in Fig. B.3 on page 250 (which is a simplified version of the example of Fig. 4.1 from page 55), the CSMA/CD protocol (inspired by [126]), a stop-and-wait protocol, implemented with one sender, one receiver and two transmission channels (inspired by [111]), and a train gate controller (inspired by [5]).

The translations into TA and TPN of the light switch example are shown in Fig. 6.31 and 6.32 respectively. These figures are screenshots of the display of the automated translations in the graphical user interfaces of UPPAAL and TINA (arranged and recoloured for the sake of readability). Note that the priority in the TPN is represented by a grey arrow instead of a dotted arrow.

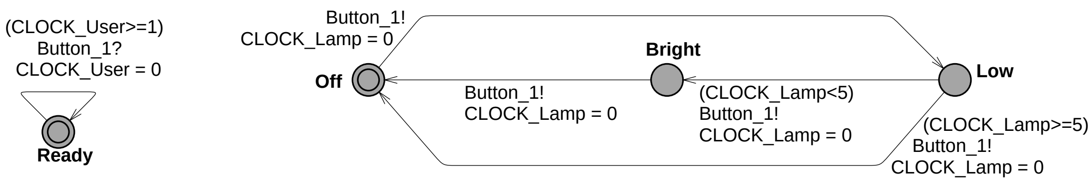

Figure 6.31: The two generated UPPAAL TA for the light switch example

Table 6.1 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[27].

The results suggest that the TA translator is efficient for programs with multiple occurrences of simple synchronizers (i.e., synchronizers involving at most two units), whereas the TPN translator is efficient for limited occurrences of more complex synchronizers.

The successful application of the tool on those benchmarks is – in addition to the remarks we made in Sections 6.1.4 and 6.2.4 – another indication that the restrictions we impose leave reasonably big translatable subsets.

Note that the current version of the translator tool does not cover the possible extensions

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

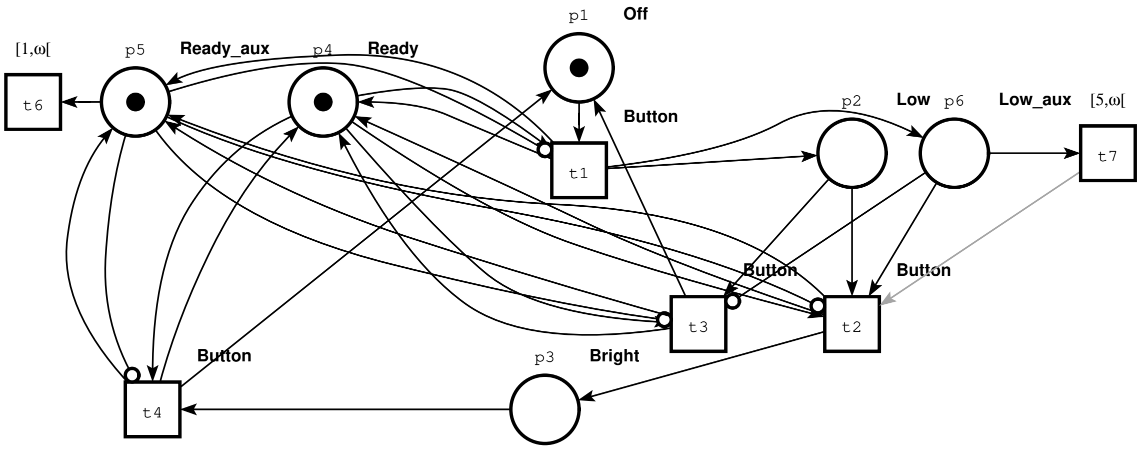

Figure 6.32: The generated TINA time Petri net for the light switch example

	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

Table 6.1: Size comparison: ATLANTIF vs. generated UPPAAL vs. generated TINA

we listed in Sections 6.1.4 and 6.2.4. In particular, the advanced multiway synchronization emulation for Uppaal is not yet implemented (and for this reason it was only sketched).

Conversely, the translator handles constructs that are not covered in this chapter, namely the possibility to import C types and functions for a translator to Tina (given that C is already Tina's language for types and functions, they are simply copied). The application example of Chapter 7 (that will also give a detailed illustration of the descriptions of this section) will use this possibility.

Chapter 7

Example: a lift

Abstract This chapter uses an example of the formal description of a lift to show how the ATLANTIF model and the atlantif tool can be used in practice. First, the lift is (manually) modelled in ATLANTIF. Then, this model is automatically translated into a TINA TPN, where formal verification is performed.

7.1 Modelling in ATLANTIF

7.1.1 The lift example

The example we discuss in this chapter is borrowed from [78], where it serves as an illustration for the language CSP-OZ-DC. Based on initial motivations similar to ours when we defined ATLANTIF, this language follows the approach of describing each different component of a system in the most suitable language: Control and communication aspects are expressed in CSP (cf. Section 3.3.2), data aspects are expressed in the *Object-Z* [53] specification language, and real-time related aspects are expressed in a subset of the *Duration Calculus* [127] logic. In the following, we will give details on these three aspects in natural language.

Control and communication. At the beginning of an execution, the lift is immobile and its control waits to be instructed by a user to move to another storey, which is represented by a communication on the gate *Newgoal*. The control reacts by starting the lift (gate *Go*), which then begins to move.

When moving, the lift cabin passes by different storeys. For each such a storey, two cases are possible: Either it is not the intended target (represented by a communication on the gate *Passed*), then the cabin continues to move; or the intended target is reached (represented by a communication on the gate *Halt*), then the lift goes back into its initial state.

Data. Three variables are used in the data part of the lift description of [78]: Two integer variables *current* and *goal* represent the current storey of the cabin and its intended target respectively[28], and the variable *dir*, which takes values in $\{-1, 0, 1\}$, represents the direction in which the cabin is currently moving (1 means "up", -1 means "down", 0 means "stationary"). Additionally, the constants *Min* and *Max* denote the lowest and the highest storey in the building respectively.

Initially, *current* and *goal* are assigned the value of *Min* (the cabin is in the lowermost basement and has not yet been instructed with a new target), and *dir* receives the value zero (the cabin does not move). During the communications that may occur, some of the variables are assigned new values, as described in the following:

- On each communication on the gate *Newgoal*, the variable *goal* receives an arbitrary value, which must be at least *Min* and at most *Max* and also different from *current* i.e., the target must be an existing storey that is not the current location of the cabin.

- On each communication on the gate *Go*, the variable *dir* receives a new value: 1 if $goal > current$, and -1 if $goal < current$ i.e., during the starting, it is calculated whether the travel goes up or down.

- On each communication on the gate *Passed*, the variable *current* receives as new value the result of "$current + dir$".

Moreover, we need the variables to satisfy the condition "$goal = current$" in order to enable a communication on the gate *Halt*.

Real time. The real-time part of the lift specification consists of two conditions that have to be satisfied by each execution of the lift model:

- The first condition is that at least three seconds[29] elapse between two consecutive communications on the gate *Passed* (the cabin needs at least three seconds to move from one storey to another).

- The second condition is that as soon as the variables *current* and *goal* happen to have the same value after having had a different value before, then the communication *Halt* must occur within two seconds (when the cabin arrives, then it must stop).

[28] Note that we suppose a European numbering of the storeys i.e., the value 0 corresponds to the ground floor. Consequently, positive numbers correspond to the upper floors, and negative numbers to the basement floors.

[29] In the remainder of this chapter, we will consider the time units to be seconds.

7.1.2 Representation in ATLANTIF

We will represent the lift in ATLANTIF by three different units: The first unit represents the *Control* of the lift (i.e., the central component), the second unit represents the *Floor_Sensor* (i.e., a component that observes the position of the cabin), and the third unit represents the *Cabin*. Note that the initial specification of [78] does not contain such a parallel composition of several processes[30], but we consider the modelling with three units to be more interesting and realistic.

We will define one synchronizer for each of the communication gates i.e., *New_Goal*, *Go*, *Halt*, and *Passed*. For technical reasons, we will also introduce another synchronizer, named *Arrival_Signal*.

Also, we note that variable sharing is not possible between three units that are active at the same time. Thus, we will declare the variables corresponding to *current*, *goal*, and *dir* locally in those units where they are necessary, and synchronize them by offers on the relevant synchronizers.

This ATLANTIF module uses functions and constants, which are not formally defined in Chapter 4. The syntax of these constructs can be found in Appendix C, and we consider them self-explanatory. We give the values 0 and 3 to the constants *Min* and *Max* respectively.

The complete specification is shown in Fig. 7.1.

In the unit *Control*, we emulate the user input for a new goal of the lift cabin by a nondeterministic assignment to the variable *Goal_C*. This is followed by a communication on *New_Goal*, where this new value is emitted as an offer. Afterwards, *Control* simply participates in the synchronizations on *Go* and on *Halt*, before again being able to generate a new value for *Goal_C*. As the value generation for *Goal_C* depends on the value of *Current_C*, the latter is updated by a reception offer during the synchronization on *Halt*. The value generation is performed by the following action:

```
select Goal_C := Current_C + 1 [] Goal_C := Current_C + 2
       [] Goal_C := Current_C + 3 end select;                .
Goal_C := Goal_C % 4;
```

where the operator "%" represents modulo, defined as in C and other programming languages. More formally, $(x \ \% \ y)$ is the smallest non-negative integer z such that z is congruent to x modulo y.

In the unit *Floor_Sensor*, the initial state *Ready* waits for either a synchronization on *New_Goal* to receive the goal in the variable *Goal_S* or for a synchronization on *Go*. In the latter case, the direction that the cabin will take is calculated by calling the function *Find_Direction*, then a jump is made to the state *Active*. The state *Active* enables two different execution paths: A synchronization on the gate *Passed* (which emits the number of the storey that is passed) leads to an update of the variable *Current_S* and a jump back to *Active*; whereas a communication on the (urgent) gate *Arrival_Signal* can only occur

[30] At least not directly: Indirectly, the Duration Calculus formulas are each translated into automata which are then composed with the control part.

```
module Lift is
dense time

constant Min: int is 0,
         Max: int is 3

function Find_Direction (Goal_F: int,
                 Current_F: int) : int is
  variables Result: int

  if (Goal_F > Current_F) then
    Result := 1
  else
    if (Goal_F < Current_F) then
      Result := -1
    else
      Result := 0
    end if
  end if

  return Result
end function

sync New_Goal is Control and Floor_Sensor
            and Cabin end sync
sync Go is Control and Floor_Sensor end sync
sync Passed is Floor_Sensor and Cabin end sync
sync Halt: urgent is Control and Floor_Sensor
            and Cabin end sync
sync Arrival_Signal: urgent is Floor_Sensor end sync

init Control,Floor_Sensor,Cabin

unit Control is
  variables Current_C:int := Min,
            Goal_C:int

  from Main
    select Goal_C := Current_C + 1
        [] Goal_C := Current_C + 2
        [] Goal_C := Current_C + 3 end select;
    Goal_C := Goal_C % 4;
    New_Goal !Goal_C; to Starting
  from Starting
    Go; to Driving
  from Driving
    Halt ?Current_C; to Main
end unit

unit Floor_Sensor is
  variables Current_S: int := Min,
            Goal_S: int,
            Direction: int

  from Ready
    select
      New_Goal ?Goal_S; to Ready
    []
      Go;
      Direction := Find_Direction(Goal_S,Current_S);
      to Active
    end select
  from Active
    select
      Passed !(Current_S + Direction);
      Current_S := Current_S + Direction;
      to Active
    []
      if (Goal_S = Current_S) then
        Arrival_Signal; to Arrived
      end if
    end select
  from Arrived
    Halt !Current_S; to Ready
end unit

unit Cabin is
  variables Current_B: int := Min,
            Goal_B: int

  from Stationary
    New_Goal ?Goal_B; to Moving
  from Moving
    select
      wait 3; Passed ?Current_B; to Moving
    []
      if (Current_B = Goal_B) then
        Halt ?any int must in [0,2]; to Stationary
      end if
    end select
end unit

end module
```

Figure 7.1: ATLANTIF code describing the lift

if *Current_S* equals *Goal_S*, in which case there is a jump to the state *Arrived* before a synchronization on *Halt* (emitting *Current_S*).

The unit *Cabin* represents all the timed constraints that we stated above. Its first discrete state is *Stationary*, from where it gets to the state *Moving* after a synchronization on *New_Goal* (also including the reception of the goal in the variable *Goal_B*). When the cabin is *Moving*, the corresponding multibranch transition assures that either three seconds elapse before each following synchronization on *Passed*, as well as that when the intended storey is reached, at most two time units elapse before it comes to a *Halt* (necessarily, because of the **must**).

7.2 Translation to TINA

Decision for TINA

It is not possible to translate the lift example to timed automata in the UPPAAL dialect, because the restrictions we stated in Section 6.1.2 exclude specifications containing an execution path with a time condition and a communication by an urgent synchronizer; they also exclude specifications containing an execution path with a data condition and a **must** communication action. Both occur in the ATLANTIF module we defined: In the unit *Cabin*, in the multibranch transition from *Moving*, the execution path which takes the second possibility in the **select** action violates the two restrictions.

It should be noted that the latter of those restrictions on our translator is not due to a limitation of the translator itself, but instead due to a limitation of the timed automata model: Strong deadlines ("**must**" communications) can only be expressed by invariant formulas in locations in the TA model, and invariants always apply to all transitions of the according location. A restriction of the invariant formula to transitions that satisfy a certain boolean condition would not be possible.

Moreover, currently only the implementation of the translation to TINA contains a prototype for the translation of functions, although it is not described in the translation definition of Section 6.2. Clearly, it would be easily possible to rewrite the module to obtain a version without a function, but this would also increase the number of execution paths in the multibranch transition from *Ready* (unit *Floor_Sensor*) and thus increase the size of the translation.

Regarding the fact that our example is timed, of course the application of our translation to FIACRE would not make sense at all. Therefore, we will continue to work on this example with a translation to TINA.

Using `atlantif`

To translate the module of Fig. 7.1 (written in a file `lift.ant`) to TINA via the `atlantif` tool, we execute the command

```
~/ATLANTIF/atlantif lift.ant -tt
```

to obtain the following output:

```
-- ATLANTIF 0.36 -- Jan Stöcker
-- (c) INRIA Grenoble Rhône-Alpes -- March 31 2009
Syntactical analysis sucessful.
Verification of the static semantics ...
done.
Translation to Tina into the file lift.net ...
Writing C code for data manipulation in file lift.c ...
done.
```

We see that the translation produces two output files, `lift.net` and `lift.c`. The first file, `lift.net`, represents the resulting time Petri net itself and is shown in Fig. 7.2.

It expresses that the generated net has twelve places, each one with an identifier from "$p1$" to "$p12$" and with a label. Three types of places can be distinguished:

- Each of the places $p1$ to $p8$ represents one of the eight discrete states of the ATLANTIF module, and are therefore labelled with the corresponding discrete state's identifier. Among them, places $p1$, $p4$, and $p7$ (representing the initial discrete states of the three units) contain initially one token each, the others being empty.
- Places $p9$ and $p10$ are auxiliary places for the place $p8$, used to express time constraints.
- Places $p11$ and $p12$ are auxiliary places for the transitions $t26$ and $t8$ respectively, used for the emptying (cf. page 162) in the translation of unit *Cabin*.

Transitions are defined by their identifier, (optionally) their label, their time interval, their in-places (left side of the arrow), their inhibitor places (left side of the arrow and with an attached "`?-1`"), and their out-places (right side of the arrow). The generated net has 13 transitions, which again fall in three different groups:

- The first seven transitions represent synchronizations (in the sense of the ATLANTIF semantics). For $t9$ and $t26$, the used synchronizers are *Arrival_Signal* and *Halt* respectively, thus their associated time interval is $[0, 0]$ (because they are **urgent**) and they are not labelled (because in ATLANTIF, urgency implies hiding). The transitions $t8$, $t4$, $t23$, $t24$, and $t25$ on the other hand are generated from visible synchronizers, thus they are labelled with the according synchronizer's name and are all defined with the time interval $[0, \infty[$.
- The two next transitions $t16$ and $t17$ represent (together with the places $p9$ and $p10$) the translation of timing constraints (corresponding to the cases 2 and 6 respectively).

```
# Automatic translation of the ATLANTIF module Lift
net Lift
pl p7 : Stationary (1)
pl p12 : Moving_out
pl p11 : Moving_out
pl p8 : Moving
pl p10 : Moving_aux
pl p9 : Moving_aux
pl p4 : Ready (1)
pl p5 : Active
pl p6 : Arrived
pl p1 : Main (1)
pl p2 : Starting
pl p3 : Driving

tr t9 [0,0] p5 -> p6
tr t26 [0,0] p10 p8 p6 p3 -> p11 p4 p1
tr t8 : Passed [0,w[ p8 p5 p9?-1 -> p12 p5
tr t4 : Go [0,w[ p4 p2 -> p5 p3
tr t23 : New_Goal [0,w[ p7 p4 p1 -> p8 p10 p9 p4 p2
tr t24 : New_Goal [0,w[ p7 p4 p1 -> p8 p10 p9 p4 p2
tr t25 : New_Goal [0,w[ p7 p4 p1 -> p8 p10 p9 p4 p2
tr t16 [3,w[ p9 ->
tr t17 [2,2] p10 ->
tr t18 [0,0] p11 -> p7
tr t19 [0,0] p9 p11 -> p7
tr t20 [0,0] p12 -> p8 p10 p9
tr t21 [0,0] p10 p12 -> p8 p10 p9

pr t19 > t18
pr t21 > t20
pr t21 t20 t19 t18 > t8 t9 t4 t23 t24 t25 t26 t16 t17
pr t26 > t17
```

Figure 7.2: The file `lift.net`

- The last four transitions are the emptying transitions following the places $p11$ and $p12$ respectively[31].

The last part of the translation describes the priorities. Again, three types of priority occur:

- The two first couples ("$t19 > t18$" and "$t21 > t20$") are part of the emptying construct, ensuring that remaining tokens in the places $p9$ and $p10$ respectively are always removed subsequent to the transitions $t26$ and $t8$ respectively.
- The next line ensures that the emptying is always finished at once when it becomes necessary, by giving emptying transitions priority over all other transitions, as explained in the context of Example 6.3.
- The last couple ("$t26 > t17$") ensures the correct translation of the **must** timing constraint, as explained in the context of Example 6.4.

The TINA toolbox provides the transformation of textual descriptions as in Fig. 7.2 into graphical representations. Therefore, we also display the graphical version of the lift TPN in Fig. 7.3 (manually arranged for better readability). Note that Fig. 7.3 does not contain the arrows representing priorities: As there are 39 of them, the net would become unclear if they were displayed.

The other output file of the translation, `lift.c`, is shown here only in extracts. It contains C code expressing the following:

- Variables: The set $\mathcal{V}_P$ i.e., all variables of the module, is represented by a single record type variable of the following form:
  ```
  typedef struct {
    int Current_S; int Goal_S; int Direction; int Current_C; int Goal_C;
    int Goal_B; int Current_B;
  } value;
  ```
  Moreover, one function defines the initial values of the variable: For *Current_C*, *Current_S*, and *Current_B*, this initial value is zero according to the module's definition. For the other variables (which do not have an initial value assigned in the module), the default value is assigned, which is also zero.
- Function: There is a C translation of the function *Find_Direction*, which is very straightforward.
- Preconditions: Two boolean functions check the following:
  - The precondition of transition $t9$: $(Goal\_S = Current\_S)$

[31] Note that the numbering of the transitions is not e.g., from $t1$ to $t13$, but seemingly much less regular. This has technical reasons, as several transitions that are generated during the translation are only temporary.

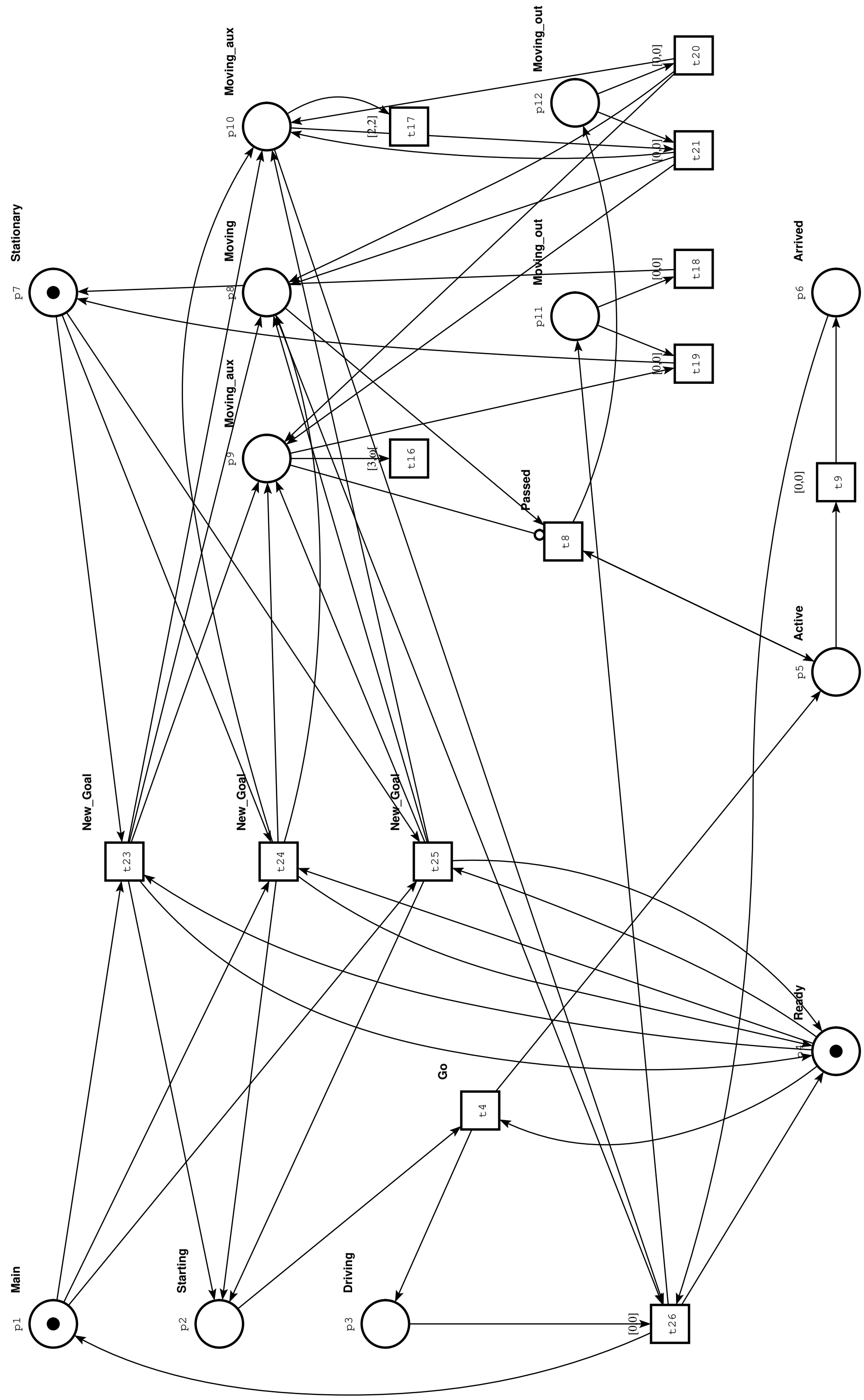

Figure 7.3: The generated TINA TPN of the lift

– The precondition of transition $t26$: ($Current\_B = Goal\_B$)

For instance, in the first case, this function takes the form

```
bool pre_0 (key s) {
  value *old = lookup(s);
  return ((old->Goal_S == old->Current_S));
}
```

,

where the data type *key* represents a record containing all variables of the model (cf. Section 6.2.1), and the function *lookup* retrieves the current values of these variables.

Note that the other transitions do not have preconditions.

- Data manipulations: Six functions, associated to the transitions $t26$, $t8$, $t4$, $t23$, $t24$, and $t25$ respectively perform modifications on the variables, each of which corresponds either to an assignment action or to a reception offer.

  For instance, in the function associated to the transition $t23$, the variable $Goal\_C$ is assigned the value $Current\_C + 3$ (assignment action in the third branch of the **select** action), then $Goal\_C$ is assigned the value $Goal\_C$ % 4 (assignment action), then $Goal\_B$ is assigned the value of $Goal\_C$ (reception offer), and then $Goal\_S$ is assigned the value of $Goal\_C$ (reception offer).

  In the generated C code, this function takes the following form:

```
key act_4 (key s) {
  value *stored = lookup(s);
  value old = {stored->Current_S, stored->Goal_S, stored->Direction,
               stored->Current_C, stored->Goal_C, stored->Goal_B,
               stored->Current_B};
  old.Goal_C = (old.Current_C + 3);
  old.Goal_C = (old.Goal_C % 4);
  old.Goal_B = old.Goal_C;
  old.Goal_S = old.Goal_C;
  return store (make_value (old.Current_S, old.Goal_S, old.Direction,
                old.Current_C, old.Goal_C, old.Goal_B, old.Current_B));
}
```

  Note that transitions other than the aforementioned six do not manipulate the variables.

As well as the elements just mentioned, the C code in the file `lift.c` also contains other, more technical functions e.g., that determine how the variable values are displayed in the output of the state space generation.

7.3 Verification

7.3.1 State space construction

A first step in the verification is the construction of the state space. The state space is a symbolic representation of the Petri net's semantics: A possibly infinite number of semantic states is grouped into one *class*, thus the state space of the Petri net can be finite, even if (which is virtually always true) the TLTS of the Petri net is infinite (cf. the semantics definition in Section 6.2.1). Each class is defined by a marking of places, by the values of the variables, and by a system of inequations that express time ranges for how long the enabled transitions have been enabled.

When performing state space generation with the command

`~/tina-2.9.2/bin/tina lift.tts`

we obtain the following information about the TPN we generated:

- It is 1-bounded e.g., no place can contain more than one token at a time.
- It contains 192 classes and 248 transitions between classes.
- All the classes are "live" i.e., there is no timelock state.
- Two of the 13 TPN-transitions ($t21$ and $t18$) are "dead" i.e., neither of them is ever fired in any possible run. Both of them are auxiliary transitions we defined for emptying, and therefore their being dead is only a technical detail. Nevertheless, this can be taken as an indication that further improvement is possible in the emptying algorithm.

7.3.2 Model checking

Model checking is used to verify whether our lift model satisfies a given property. As an example, we will check the property "Each time the lift is assigned a new target, it eventually stops afterwards.".

The TINA toolbox contains the tool `selt` [22], which performs verifications on formulas given in a variation of LTL (linear temporal logic). In this logic, our property takes the form

"`[] ((`$t23$ `\/` $t24$ `\/` $t25$`) => (<>` $t26$`))`",

where this fomula is composed as follows:

- $t23$, $t24$, and $t25$ are those transitions corresponding to a synchronization on the gate *New_Goal*. The term `(`$t23$ `\/` $t24$ `\/` $t25$`)` expresses a disjunction between them.
- $t26$ is the only transition corresponding to a synchronization on *Halt*. The term `(<>` $t26$`)` expresses "eventually $t26$ occurs".
- Then, the entire formula can be read as "On every path containing a synchronization on *New_Goal*, afterwards there will be eventually a synchronization on *Halt*.".

Clearly, it is a drawback that we need to read the file `lift.net` in order to find the identifiers of those transitions labelled *New_Goal* and *Halt* respectively, but the current version of `selt` does not support a high-level syntax that e.g., identifies transitions by their labels.

We can check the formula by executing

```
> ~/tina-2.9.2/bin/tina lift.tts -ktz > lift.ktz
> ~/tina-2.9.2/bin/selt lift.ktz -f "[] ((t23 \/ t24 \/ t25) => (<> t26))"
```

and then obtain the result

```
Selt version 2.9.2 -- 05/02/08 -- LAAS/CNRS
ktz loaded, 192 states, 248 transitions
0.004s
FALSE
  state 0: L.div p1 p4 p7
  -t25 ... (preserving (- t26 /\ t23) \/ (- t26 /\ t25) \/ - t26 /\ t24)->
  state 1: p10 p2 p4 p8 p9 Goal_S Goal_C Goal_B
  -t17 ... (preserving - t26)->
* [accepting] state 4: L.div p3 p5 p8 Goal_S Direction Goal_C Goal_B
  -time-divergence ... (preserving - t26)->
  state 4: L.div p3 p5 p8 Goal_S Direction Goal_C Goal_B
0.002s
```

Thus, our lift model does not satisfy the property. The `selt` tool gives us the prefix of a run that constitutes a counter-example for the property. Reading this prefix, we see that from the initial state, the goal "storey one" is generated, then two seconds elapse, then the auxiliary transition $t17$ reaches its limit and fires, and then time elapses indefinitely without the firing of any transition.

Intuitively, we would suppose the lift to start (synchronizer "*Go*"), but as we can see in the module of Fig. 7.1, there is no restriction for the elapsing of time at this configuration, thus it is *possible* that the lift does not start. When looking further in Fig. 7.1, we see a second, very similar, problem in the module: Before a synchronization on *Passed*, at least three seconds must elapse, but there is no limit on this elapsing. Thus, another infinite time elapsing is possible when a synchronization on *Passed* is expected i.e., when the cabin moves from one storey to the next.

We can correct these two problems for instance by the following modifications to the lift module:

- We declare the synchronizer *Go* to be **urgent** i.e., after the choice of a new goal, the cabin starts to move at once.
- We modify the first branch of the **select** action of the multibranch transition of the discrete state *Moving* in the unit *Cabin* from

  wait 3; *Passed* ?*Current_B*; **to** *Moving*

  to

  wait 3; *Passed* ?*Current_B* **must in** [0,1]; **to** *Moving*

  i.e., we now describe that the Cabin needs at least three, but at most four seconds to get from one storey to the next.

When we translate the modified lift module to TINA and verify the corresponding LTL formula (modulo a renumbering of the transitions), we get the following results:

```
> ~/tina-2.9.2/bin/selt lift2.ktz -f "((t31 \/ t32 \/ t33) => (<> t34))"
Selt version 2.9.2 -- 05/02/08 -- LAAS/CNRS
ktz loaded, 138 states, 152 transitions
0.005s
TRUE
0.002s
```

The ATLANTIF model with these modifications now satisfies the property expressed by the formula.

7.4 Conclusion

In this chapter, we showed an example of the steps in using the ATLANTIF model and the `atlantif` tool. In particular, we illustrated how the automatic translation tool can be used to perform model-checking on an ATLANTIF module; and then to debug this module. Although we only made a translation to the time Petri net dialect of TINA (because the given ATLANTIF code only enabled to use the TINA translator), the basic approach would be the same for other target languages.

Secondary, we demonstrated that the expressive power of ATLANTIF is not limited to represent constructs from languages from the family of LOTOS, which were dominating Chapter 5, but our modelling in this chapter also indicated that, at least partially, a very different model approach such as CSP-OZ-DC is also covered.

Also, we are able to compare the approach of CSP-OZ-DC to the ATLANTIF approach, which is indeed interesting, because both share similar motivations. In our opinion, ATLANTIF is more intuitive than CSP-OZ-DC, because of its more textual syntax, which is easier to read and to write. Moreover, the user only has to know a single language instead of three.

Chapter 8

Conclusion

8.1 Contribution

In this thesis, we introduced the new formal model Atlantif, designed to provide rich expressive power to represent realistic systems, and situated as an intermediate model between high-level languages and graphical models. Based on an analysis of existing languages and models, we formally defined and discussed the syntax and the semantics (expressed in the form of a Tlts) of our model.

Furthermore, we showed the suitability of Atlantif as an intermediate format: We illustrated the expressive power of Atlantif by representing in our model typical high-level constructs, and we provided translations to two graphical models.

8.1.1 Language features

In Chapter 1, we developed the objective to define an intermediate model with a rich expressive power regarding data handling, concurrency, and real time, which is met by the Atlantif model in the following way:

- Regarding *data handling*, Atlantif (as well as its structure of sequential processes) is strongly based on the successful Ntif language. We did not need to define extensions for this aspect, because Ntif already can successfully represent high-level data handling.

- Regarding *concurrency*, Atlantif features the notion of synchronizers, which allows in a concise, powerful, and intuitive way the representation of possible synchronizations between units that are composed in parallel.

  A synchronization generally yields a transition in the underlying Tlts, unless the corresponding synchronizer was declared as **silent**. Silent synchronizers enable the representation of powerful high-level constructs such as exception handling, while preservaing their semantics.

- Regarding *real time*, ATLANTIF introduces an independent delay action and a life reducer extension to the communication action, which both enable to control time elapsing and occur similarly in different existing high-level languages. ATLANTIF also introduces the possibility to declare a synchronizer to be **urgent**. These constructs enable both weak and strong deadlines.

  In the formal semantics definition, we introduced the new notion of the phase as a key element to easily ensure the satisfaction of generally accepted good properties of the semantics: time additivity, time determinism, and maximal progress of urgent actions.

We defined a formal semantics for ATLANTIF in Chapter 4, using elaborate rules, which highlight the consequences of combining data, real time, and concurrency in one model: For instance, the distinction of several cases in the recalculation of a phase after a synchronization (predicate $next\_\theta$ on page 84) is due to the combination of real time and concurrency. Also, it can be observed that small simplifications of the syntax lead to a considerable simplification of the semantics [118].

8.1.2 Comparison of ATLANTIF with related work

Several intermediate models have been defined with objectives similar to ATLANTIF's. In Section 3.4, we gave an overview describing the central ideas and characteristics of these models. We now compare them with ATLANTIF systematically.

Table 8.1 indicates the differences between ATLANTIF and the other intermediate models with respect to a few criteria discussed in Section 3.5.1 and briefly recalled them in the following:

- "*time domain*" indicates whether the model accepts discrete, dense, or both time domains (cf. Definition 3.2).
- "*weak/strong deadlines*" indicates whether the model can express weak deadlines, strong deadlines, or both (cf. page 46).
- "*clocks*" indicates whether the model uses clocks like those in timed automata (cf. Definition 3.7) or more abstract high-level constructs.
- "*life reducers*" indicates whether a communication can be associated with a construct that describes the time instants at which the communication may happen.
- "*delay constructs*" indicates whether the model contains an explicit construct (such as the **wait** action of ATLANTIF) that enables time to elapse independently of any communication.
- "*global variables*" indicates whether the model provides variables that can be accessed concurrently by different processes. This excludes models allowing a limited sharing of local variables such as ALTARICA and ATLANTIF. For instance, in ATLANTIF, a

	time domain		deadlines								synchronization				
	dense	discrete	weak	strong	clocks	life reducers	delay constructs	global variables	hybrid variables	high-lvl trans.	simple	complex	start/stop	priorities	probabilities
IF-2.0	+	−	+	+	−	+	−	+	−	−	−	−	+	(+)	−
ALTARICA	+	−	+	+	+	+	−	−	−	−	+	+	−	+	−
NTIF	−	−						−	−	+	−	−	−	−	−
FIACRE	+	−	−	+	−	+	−	+	−	+	+	+	−	+	−
BIP	−	+	+	+	+	+	−	−	+	−	+	+	−	+	−
MODEST	+	−	+	+	+	+	−	+	−	+	+	−	−	−	+
ATLANTIF	+	+	+	+	−	+	+	−	−	+	+	+	+	−	−

Table 8.1: Comparison of intermediate models

variable can be accessed in different subunits, but the static restrictions ensure that a process can only write this variable if no other process that may read this variable is running.

- "*hybrid variables*" indicates whether the model provides variables that evolve linearly or non-linearly with the elapsing of time (cf. Section 3.2.2).

- "*high-level transitions*" indicates whether the model provides expressive and concise notations such as the multibranch transitions of NTIF, as opposed to the approach of "condition/action"-transitions (cf. Section 3.4.6).

- "*simple synchronization*" indicates whether the model provides gate synchornization.

- "*complex synchronization*" indicates whether the model provides for a given state and a given gate a choice between different synchronization possibilities, as opposed to simple sets of processes that can synchronize (for instance, containing either two processes or all processes).

- "*start/stop*" indicates whether the model provides dynamic starting and stopping of processes.

- "*priorities*" indicates whether the model provides priorities between the gates on which the concurrent processes synchronize. In the case of IF-2.0 (which does not provide synchronizations), a process can have priority for discrete actions over other processes.

- "*probabilities*" indicates whether the model provides a choice that makes some alternatives more likely than others.

In addition to the schematic overview of Table 8.1, we discuss the main differences between Atlantif and Bip, MoDeST, and Fiacre.

- Fiacre and Atlantif both use multibranch transitions, a notion inherited from Ntif. Therefore, both models are very similar as regards processes (respectively units), with the exception of real-time constructs: Atlantif units provide delays and life reducers, while Fiacre processes do not provide real-time constructs. Instead, Fiacre provides life reduces associated with gates. Most high-level languages also provide real-time constructs within processes, which favors Atlantif as an intermediate format.

  Moreover, Atlantif provides real-time constructs both with strong and with weak deadlines, whereas Fiacre only provides strong deadlines.

  Also, Fiacre and Atlantif differ in the notation of synchronization: Fiacre uses a variant of synchronization vectors that combine several process identifiers and several gate identifiers within a single construct; Atlantif associates with each gate exactly one synchronization formula. We consider these separated formulas of the Atlantif approach to be a more concise and intuitive notation. Additionally, only Atlantif provides dynamic starting and stopping of units.

  Other than Atlantif, Fiacre features priorities between gates, which we considered not to be essential for an intermediate format.

- Bip and Atlantif both provide original constructs for complex gate synchronizations. However, the expressive power of these constructs is different, as Bip does not provide dynamic starting and stopping of units. On the other hand, Bip provides a broadcast communication construct, which is not included in Atlantif, because it would introduce unintuitive and complex semantic rules.

  An Atlantif unit provides the high-level syntax of multibranch transitions, whereas Bip provides simple condition/action transitions. Therefore, Bip processes are less concise and further from high-level languages than Atlantif units.

  Other than Atlantif, Bip features priorities and hybrid variables. We consider the latter to be not essential for modelling and verification of those systems for which we designed Atlantif.

- MoDeST and Atlantif both provide process descriptions using high-level constructs, although Atlantif structures units by discrete states, which do not exist in MoDeST. To describe complex processes, discrete states can simplify readability and modelling. Also, time in MoDeST processes is implemented by clock variables, whereas ATLANTIF uses high-level constructs (**wait** and life reducer in communication).

The concept of different synchronization sets provided by ATLANTIF does not exist in the parallel composition operator of MoDeST, nor the possibility to start and stop processes.

Other than ATLANTIF, MoDeST extends the nondeterministic choice with probabilities, which again is an aspect we do not consider to be essential for the objectives of ATLANTIF.

In summary, we can see firstly that the expressive power of ATLANTIF is not covered by the existing intermediate models, and secondly that ATLANTIF provides a syntax that is particularily close to high-level languages.

8.1.3 Extension of the possibilities to use formal verification

Formal verification of reasonably big subsets of ATLANTIF can be done using the translator tool presented in Chapter 6. The translations required to define emulations for some constructs of ATLANTIF that have no direct correspondence in the target language, most importantly the following:

- For UPPAAL, we emulated multiway synchronization, unit stopping and starting, and offers.
- For TINA, we emulated weak deadlines and time constraints on the process level.

The different translators cover different subsets of ATLANTIF, where neither is a subset of the other. In practice, this enables us in many cases (as observed in Sections 6.1.4, 6.2.4, 7.2, and in Appendix A.3.5) to find a suitable translator for a given specification. Therefore, the verifiable subset of ATLANTIF is considerable.

The central intention of this thesis was to bring closer the two tasks of expressive and concise modelling and formal verification, with the objective to extend of the class of systems that can be verified in practice. We consider this goal to be achieved, for the following reasons:

- Compared to the high-level languages discussed in Section 3.3, ATLANTIF extends the possibilities for verification, because it combines elements of high-level syntax with a link to verification tools. Such a link is rarely provided for high-level languages, at least for those that combine concurrency with real time and data handling.
- Compared to other intermediate models, ATLANTIF extends the possibilities for verification, because, as we observed in Section 8.1.2, our model complements the other models by proposing a new and unique combination of syntax features. While it should be noted that ATLANTIF has neither a strictly larger nor a strictly smaller

expressive power than other intermediate models[32], it is designed to be close to powerful process algebras such as E-Lotos and Lotos NT, and therefore a valuable complement.

- Compared to graphical models, Atlantif extends the possibilities for verification, because it provides concise constructs (such as synchronizers, offers, and multibranch transitions) that enable users to easily describe systems that would be difficult to describe directly in a graphical model. Using one of our translations, formal verification is then possible.

  Note that obviously the subsets of Atlantif that can be translated to graphical models are not more expressive than the graphical models themselves. But for instance, an Uppaal user wanting to design a system with multiway synchronization or a Tina user wanting to design a system with weak deadlines will not easily find the correct representation in these models. In Atlantif on the other hand, the user can easily specify those aspects and translate to the desired model automatically.

8.2 Perspectives

8.2.1 Advancements of ATLANTIF

Syntax and semantics extensions. In Section 4.8.2, we discussed several extensions that could be brought to Atlantif. The comparison with other models discussed in Section 8.1.2 indicates that the introduction of a broadcast communication or of priorities might make sense.

Another possible extension of Atlantif lies in the formal definition of syntax and semantics (static and dynamic) for types, functions, and constants. The tool implementation provides limited support for these constructs (as seen in Chapter 7), yet formal definitions are still lacking.

Syntax modifications. Besides strict extensions, also modifications could be considered. In particular, this concerns the possibility of *dynamic instantiation* of units i.e., the definition of units in the form of templates (possibly parameterized), where several copies of such units could be active at the beginning of or during an execution. This approach exists in the If-2.0 model.

The introduction of instantiation in Atlantif would create new and powerful usages of the starting and stopping of units and enable a simple representation of the process instantiation that exists in most high-level languages. However, to our knowledge neither Uppaal nor any other tool based on timed automata supports dynamic instantiation, thus a translation of this feature would not be possible; a translation to time Petri nets would also be limited.

[32]Note that this statement still holds relative to the subsets of Atlantif that can be translated to verification tools.

Therefore, it could make sense to introduce a *static* instantiation in the ATLANTIF syntax i.e., all instances are defined before the execution of the model. This approach exists e.g., in the intermediate models BIP and FIACRE, as well as in the UPPAAL tool. Such a modification would enable a shorter syntax in some cases (e.g., in Fig. A.14, where *B1*, *B2*, and *B3* could be instances of the same unit), but the readability might be reduced.

Another interesting idea could be to introduce a graphical notation for the ATLANTIF syntax to make it more intuitive.

8.2.2 Extension of the translations

In Sections 6.1.4, 6.2.4, and in Appendix A.3.5, we discuss how the existing translations could be extended. More generally, it would be interesting to provide, even in a limited way, a correct translation for **silent** synchronizers. Especially in those cases where no loops of silent synchronizations are possible, a solution should exist.

Another direction that could be taken for an extension of the translation tool would be to define a function that automatically checks which of the translations are possible in view of the translation restrictions. In a second step, such a function could even do an analysis of which target language (if several are possible) would be the most adapted for a given model e.g., it would recommend UPPAAL if many synchronizers occur often within one unit, and TINA if several multiway synchronizations occur.

Finally, the translation could be extended to other dialects of timed automata (cf. Section 6.1.1) and of time Petri nets (cf. Section 6.2.1), or even to other models. Such new translations would make sense if they extend the subsets of ATLANTIF on which formal verification could be performed.

8.2.3 Development of FIACRE

One intention in the development of ATLANTIF was to provide ideas on how the FIACRE model could evolve in the future. The FIACRE model has been defined very recently[33], and it is possible that it will be revised in the future.

This idea is interesting, because both ATLANTIF and FIACRE are based on NTIF, and both define extensions for real time and concurrency. Regarding the possible approaches for such extensions (cf. Section 3.5.1), several decisions have been made in different ways (not intentionally, though), causing fundamental differences as described in Section 8.1.2.

For instance, the introduction of timed constructs in processes could be one ATLANTIF-inspired extension to FIACRE. Such an extension would not come at the price of reduced possibilities for formal verification, because timed verfication on FIACRE programs is currently implemented by a translation to the TINA dialect of time Petri nets, using the "`frac`" tool, and in Section 6.2 we showed that the timed constructs of ATLANTIF can also be translated to the TINA model. At the same time, new modelling opportunities in

[33]The first publications on FIACRE [24, 25] were in 2008.

FIACRE would be established.

8.2.4 Using ATLANTIF on more complex specifications

An important objective in the further development of ATLANTIF is to generalize, at least partially, the translations from high-level languages to our model. For many high-level constructs such as those discussed in Chapter 6, a more general and systematic translation should be possible, similar to the definitions already provided in Section 5.5.

Such a generalization would enable us to use ATLANTIF on more complex specifications than for instance the lift example we discussed in Chapter 7, and to make use of the advantages we discussed in Section 8.1.3.

Bibliography

[1] W. Aalst. Interval Timed Coloured Petri Nets and their Analysis. In *Application and Theory of Petri Nets*, volume 691, pages 453–472. Springer-Verlag, 1993.

[2] T. Agerwala. A Complete Model for Representing the Coordination of Asynchronous Processes. Hopkins Computer Research Report 32, Johns Hopkins University, Baltimore, July 1974.

[3] R. Alur, C. Courcoubetis, and D. L. Dill. Model-Checking for Real-Time Systems. In *LICS*, pages 414–425, 1990.

[4] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proc. of the 17th International Colloquium on Automata, Languages, and Programming ICALP*, volume 443 of *LNCS*, pages 322–335. Springer-Verlag, 1990.

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

[6] 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.

[7] André Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Masson, 1992.

[8] E. Asarin, O. Maler, and A. Pnueli. Reachability analysis of dynamical systems having piecewise-constant derivatives. *Theoretical Computer Science*, 138(1):35–65, 1995.

[9] J. Baeten and J. Bergstra. Real time process algebra. *Journal of Formal Aspects of Computing Science*, 3(2):142–188, 1991.

[10] 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.

[11] 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.

[12] G. Behrmann, A. David, and K. Larsen. *A Tutorial on Uppaal*, 2004.

[13] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Computation*, 60:109–137, 1984.

[14] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[15] B. Berthomieu, J. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stöcker, and F. Vernadat. The Syntax and Semantics of Fiacre – version 2.0. Project deliverable F3.2.2 (updated), AESE (*pôle de compétitivité mondial Midi-Pyrénées & Aquitaine: Aéronautique, Espace et Systèmes Embarqués*) project Topcased, April 2007.

[16] 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.

[17] 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.

[18] B. Berthomieu, D. Lime, O. Roux, and F. Vernadat. Reachability Problems and Abstract State Spaces for Time Petri Nets with Stopwatches. To appear, 2007, August 2006.

[19] B. Berthomieu and M. Menasche. An enumerative Approach for analyzing Time Petri nets. In *IFIP*, 1983.

[20] 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.

[21] B. Berthomieu, P.O. Ribet, F. Vernadat, J. Bernartt, J.-M. Farines, J.-P. Bodeveix, M. Filali, G. Padiou, P. Michel, P. Farail, P. Gaufillet, P. Dissaux, and J.-L. Lambert. Towards the verification of real-time systems in avionics: the COTRE approach. In Thomas Arts and Wan Fokkink, editors, *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2003 (Trondheim, Norway)*, volume 80 of *Electronic Notes on Theoretical Computer Science*, pages 201–216. Elsevier, June 2003.

[22] B. Berthomieu and F. Vernadat. Réseaux de Petri temporels : méthodes d'analyse et vérification avec TINA, 2006. Ecole d'été temps réel , Nancy. Traité IC2 : Systèmes temps réel 1 – techniques de description et de vérification.

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

[24] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frédéric Lang, and François Vernadat. FIACRE: an Intermediate Language for Model Verification in the TOPCASED Environment. In Jean-Claude Laprie, editor, *Proceedings of the 4th European Congress on Embedded Real-Time Software ERTS'08 (Toulouse, France)*. SIA (the French Society of Automobile Engineers), AAAF (the French Society of Aeronautic and Aerospace), and SEE (the French Society for Electricity, Electronics, and Information & Communication Technologies), January 2008.

[25] Bernard Berthomieu, Hubert Garavel, Frédéric Lang, and François Vernadat. Verifying Dynamic Properties of Industrial Critical Systems Using TOPCASED/FIACRE. *ERCIM News*, 75:32–33, October 2008.

[26] D. Beyer. *Formale Verifikation von Realzeit-Systemen mittels Cottbus Timed Automata*. PhD thesis, BTU Cottbus, 2002.

[27] D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A Tool for BDD-based Verification of Real-Time Systems. In *15th International Conference on Computer Aided Verification CAV 2003*, volume LNCS 2725, pages 122–125. Springer-Verlag, 2003.

[28] N. Bjørner, Z. Manna, H. Sipma, and T. Uribe. Deductive Verification of Real-time Systems Using STeP. *Theoretical Computer Science*, 253:27–60, 2001.

[29] 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.

[30] Henrik Bohnenkamp, Pedro R. d'Argenio, Holger Hermanns, and Joost-Pieter Katoen. MoDeST: A Compositional Modeling Formalism for Real-Time and Stochastic Systems. *IEEE Transactions on Software Engineering*, 32(10):812–830, 2006.

[31] 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.

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

[33] 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.

[34] Pierre Boullier and Philippe Deschamp. Le système SYNTAX : Manuel d'utilisation et de mise en œuvre sous Unix. `http://www-rocq.inria.fr/oscar/www/syntax`, October 1997.

[35] D. Bošnački and D. Dams. Integrating Real Time into Spin: A Prototype Implementation. In *FORTE/PSTV XVIII*, pages 423–439. Kluwer Academic Publishers, 1998.

[36] 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.

[37] M. Bozga, S. Graf, and L. Mounier. Automated validation of distributed software using the IF environment. In S. Stoller and W. Visser, editors, *Workshop on Software Model-Checking, associated with CAV'01*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, July 2001.

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

[39] M. Bozga, O. Maler, and S. Tripakis. Efficient Verification of Timed Automata Using Dense and Discrete Time Semantics. In *Conference on Correct Hardware Design and Verification Methods*, pages 125–141, December 1998.

[40] M. Broy. Formal Description Techniques - How Formal and Descriptive are they? In *FORTE*, pages 95–110, 1996.

[41] J. A. Brzozowski and C-J. H. Seger. Advances in Asynchronous Circuit Theory - Part II: Bounded Inertial Delay Models, MOS Circuits, Design Techniques. *Bulletin of the European Association for Theoretical Computer Science*, 43:199–263, February 1991.

[42] G. Bucci, L. Sassoli, and E. Vicario. Oris: a tool for state space analysis of real-time preemptive systems. In *1st International Conference on the Quantitative Evaluation of Systems (QEST) 2004*, 2004.

[43] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: A Tool for Quantitative Analysis of Finite-State Real-Time Systems. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[44] F. Cassez and F. Laroussinie. Model-Checking for Hybrid Systems by Quotienting and Constraints Solving. In A. Emerson and P. Sistla, editors, *12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 373–388, Chicago, Illinois, USA, July 2000. Springer-Verlag.

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

[46] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[47] 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.

[48] J.-P. Courtiat, C. A. S. Santos, C. Lohr, and B. Outtaj. Experience with RT-LOTOS, a temporal extension of the LOTOS formal description technique. *Computer Communications*, 23(12), 2000.

[49] P. D'Argenio and E. Brinksma. A Calculus for Timed Automata. In *FTRTFT*, 1996.

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

[51] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In *Hybrid Systems III, Verification and Control*, pages 208–219, 1996.

[52] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with Kronos. In *7th IFIP WG G.1 International Conference of Formal Description Techniques FORTE'94*, pages 227–242, 1994.

[53] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards & Interfaces*, 17(5-6):511–533, 1995.

[54] A. Einstein. Zur Elektrodynamik bewegter Körper. *Annalen der Physik*, 17(891), 1905.

[55] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Computer-Aided Verification (CAV'90)*, volume 531 of *LNCS*, pages 136–145. Springer-Verlag, 1990.

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

[57] P. Feiler, D. Gluch, and J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Note CMU/SEI-2006-TN-011, Carnegie Mellon, 2 2006.

[58] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

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

[60] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[67] G. Gössler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005.

[68] J. F. Groote. Transition system specifications with negative premises. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, number 458 in LNCS, pages 332–341. Springer-Verlag, 1990.

[69] J. F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[70] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The Formal Specification Language mCRL2. In *Methods for Modelling Software Systems. Dagstuhl Seminar Proceedings*, 2007.

[71] J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes'94*, Workshops in Computing Series, pages 26–62. Springer Verlag, 1995.

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

[73] T. Henzinger. The Theory of Hybrid Automata. *Verifications of Digital and Hybrid Systems*, 170:265–292, 2000.

[74] T. Henzinger, P. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[75] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, pages 193–244, June 1994.

[76] Christian Hernalsteen. *Specification, Validation and Verification of Real-Time Systems using ET-LOTOS*. Thèse de Doctorat, Université Libre de Bruxelles, June 1998.

[77] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[78] J. Hoenicke and P. Maier. Model-Checking of Specifications Integrating Data, Processes and Time. In J. S. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005*, volume 3582 of *LNCS*, pages 465–480. Springer Verlag, 2005.

[79] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Software Series. Prentice Hall, 1991.

[80] P. Hsiung, Y. Chen, and Y. Lin. Model Checking Safety-Critical Systems Using Safecharts. *IEEE Transactions on Computers*, 56(5):692–705, 2007.

[81] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[82] ISO/IEC. Syntactic metalanguage — Extended BNF. International Standard 14977:1996(E), International Organization for Standardization — Information Technology, Genève, December 1996.

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

[84] 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.

[85] A. Klusener. *Models and axioms for a fragment of real time process algebra*. PhD thesis, TU Eindhoven, December 1993.

[86] Frédéric Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer Verlag, November 2005. Full version available as INRIA Research Report RR-5673.

[87] F. Laroussinie and K. Larsen. Compositional Model-Checking of Real Time Systems. In *CONCUR'95, Philadelphia, USA*, volume 962 of *LNCS*, pages 27–41. Springer-Verlag, August 1995.

[88] F. Laroussinie and K. Larsen. CMC: A Tool for Compositional Model-Checking of Real-Time Systems. In *FORTE XI*, pages 439–456, Deventer, The Netherlands, 1998. Kluwer, B.V.

[89] 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.

[90] L. Léonard. *An Extended LOTOS for the Design of Time-Sensitive Systems*. PhD thesis, Université de Liège, 1997.

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

[92] Luc Léonard and Guy Leduc. An Enhanced Version of Timed LOTOS and its Application to a Case Study. In Richard L. Tenney, Paul D. Amer, and M. Umit Uyar, editors, *Proceedings of the 6th International Conference on Formal Description Techniques FORTE'93 (Boston, MA, USA)*, pages 483–498. North-Holland, October 1993.

[93] D. Lime, O. Roux, C. Seidner, and L. Traonouez. Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches. In S. Kowalewski and A. Philippou, editors, *TACAS 2009*, volume 5505 of *LNCS*, pages 54–57. Springer Verlag, 2009.

[94] C. Lohr and J.-P. Courtiat, 2002. From the specification to the scheduling of time-dependent systems.

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

[96] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[97] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[98] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90 (Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, pages 401–415. Springer-Verlag, August 1990.

[99] 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.

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

[101] D. C. Oppen. A $2^{2^{2^{pn}}}$ Upper Bound on the Complexity of Presburger Arithmetic. *Journal of Computer and System Sciences*, 16:323–332, 1978.

[102] J. S. Ostroff. Verification of safety critical systems using TTM/RTTL. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 573–602. Springer-Verlag, June 1991.

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

[104] M. Ouimet and K. Lundqvist. Verifying Execution Time using the TASM Toolset and UPPAAL. Technical Report ESL-TIK-00212, MIT, 2008.

[105] G. D. Plotkin. A Structural Approach to Operational Semantics. Lecture Notes DAIMI FN-19, Aarhus University, 1981.

[106] C. Ramchandani. Analysis of Asynchronous Concurrent Systems by Timed Petri Nets. Technical Report TR-120, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[107] A. Ratzer, L. Wells, H. Lassen, M. Laursen, J. Qvortrup, M. Stissig, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Net. In *24th International Conference of Applications and Theory of Petri Nets*, volume 2679 of *LNCS*, pages 450–462, 2003.

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

[109] S. Roch and P. Starke. *INA – Integrated Net Analyzer – Version 2.2 – Manual.* Humbold-Universität Berlin, April 1999.

[110] J. Ruf and T. Kropf. Analyzing Real-Time Systems. In *Conference on Design Automation and Test in Europe*. IEEE Computer Society Press, March 2000.

[111] T. Sadani. *Vers l'utilisation des réseaux de Petri temporels étendus pour la vérification de systèmes temps-réel décrits en RT-LOTOS.* PhD thesis, INP Toulouse, 2007.

[112] T. Sadani, M. Boyer, P. Saqui-Sannes, and J.-P. Courtiat. Effective Representation of RT-LOTOS Terms by Finite Time Petri Nets. In *Formal Techniques for Networked and Distributed Systems – FORTE 2006*, pages 404–419, September 2006.

[113] L. Sassoli and E. Vicario. Analysis of real time systems through the ORIS tool. In *Quantitative Evaluation of Systems (QEST06)*, Riverside, sep 2006. IEEE Computer Society Press.

[114] Philippe Schnoebelen. Refined Compilation of Pattern-Matching for Functional Languages. *Science of Computer Programming*, 11:133–159, 1988.

[115] Mihaela Sighireanu. *Contribution à la définition et à l'implémentation du langage "Extended LOTOS".* Thèse de Doctorat, Université Joseph Fourier (Grenoble), January 1999.

[116] Mihaela Sighireanu. LOTOS NT User's Manual (Version 2.6). INRIA project-team VASY. `ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z`, February 2008.

[117] P. Starke and S. Roch. Ina et al. In Kjeld H. Mortensen, editor, *Tool Demonstrations 21st International Conference on Application and Theory of Petri Nets*, pages 51–56, June 2000.

[118] J. Stöcker, F. Lang, and H. Garavel. Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format. In M. Leuschel and H. Wehrheim, editors, *Proc. of the 7th International Conference on integrated Formal Methods iFM*, number 5423 in LNCS, pages 88–102. Springer-Verlag, February 2009. Shorter, therefore incomplete version of [119].

[119] J. Stöcker, F. Lang, and H. Garavel. Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format. Research Report RR-6950, INRIA Grenoble - Rhône-Alpes, June 2009. Extended (complete) version of [118].

[120] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Syntax and Semantics of Timed Chi. Technical Report 05-09, TU Eindhoven, 2005.

[121] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Syntax and Consistent Equation Semantics of Hybrid Chi. *Journal of Logic and Algebraic Programming, special issue on hybrid systems*, 68(1–2), 2006.

[122] A. Verdejo. E-LOTOS Tutorial with Examples, April 2000.

[123] 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.

[124] M. Weber. An Embeddable Virtual Machine for State Space Generation. In D. Bošnački and S. Edelkamp, editors, *14th International SPIN Workshop*, volume 4595 of *LNCS*, pages 168–185. Springer-Verlag, 2007.

[125] 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.

[126] 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.

[127] C. Zhou, C. Hoare, and A. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

Appendix A

Additional algorithms and proofs

A.1 Static semantics

A.1.1 Unicity of communication, undelayed next state reachability

In Section 4.4.3, we introduced the static semantics constraint that each execution path in a multibranch transition may only contain a single communication, and that each execution path with a communication, this communication must be followed by a jump or a **stop**, while it must not be followed by a delay. A similar criterion (without the restriction on the delay and without the possibility of a **stop**) has already been formally defined in Appendix A.4 of [61]; in this section we will extend it to the full ATLANTIF syntax.

A discrete state "**from** s A" satisfies the unicity of communication and undelayed next state reachability if $ucuns(A)$ equals **true**, where $ucuns$ is a boolean function defined in Fig. A.1.

Note that the function *reach* uses the function *exh* to check the case exhaustivity of the patterns of a **case** action, which is necessary to ensure the next state reachability after a communication. The function *exh* is already formally defined in Appendix A.5 of [61] and shall not be repeated here.

Clearly, this formal definition does not provide a necessary criterion, as it also excludes actions such as

$$\textbf{while false do } G \textbf{ end}; \ \textbf{to } s',$$

which does not allow an execution path that violates the unicity of communication and undelayed next state reachability as we defined it intuitively. However, only exotic cases such as this example seem to be excluded.

$$
\mathit{ucuns}(A) = \begin{cases}
\textbf{true} & \text{if } A \text{ has the form } \textbf{null},\ \textbf{to}\ s',\ \textbf{wait}\ E,\ \textbf{stop},\ \textbf{to}\ s'\textbf{;}\ A_1, \\
& \text{or } \textbf{stop;}\ A_1 \\
\mathit{ucuns}(A_1) & \text{if } A \text{ has the form } A_0\textbf{;}\ A_1 \text{ where } A_0 \text{ has the form} \\
& V_0\textbf{,}\dots\textbf{,}V_n\ \texttt{:=}\ E_0\textbf{,}\dots\textbf{,}E_n, \\
& V_0\textbf{,}\dots\textbf{,}V_n\ \texttt{:=}\ \textbf{any}\ T_0\textbf{,}\dots\textbf{,}T_n\ \textbf{where}\ E, \\
& \textbf{reset}\ V_0\textbf{,}\dots\textbf{,}V_n,\ \textbf{wait}\ E, \text{ or } \textbf{null} \\
\mathit{reach}(A_1) & \text{if } A \text{ has the form } G\ O_1 \dots O_n\ Q\ \textbf{in}\ W\textbf{;}\ A_1 \\
\mathit{ucuns}(A_1\textbf{;}\ A_3) \wedge \mathit{ucuns}(A_2\textbf{;}\ A_3) & \text{if } A \text{ has the form } \textbf{if}\ E\ \textbf{then}\ A_1\ \textbf{else}\ A_2\ \textbf{end;}\ A_3 \\
\bigwedge_{i\in 0..n} \mathit{ucuns}(A_i\textbf{;}\ A') & \text{if } A \text{ has the form } \textbf{select}\ A_0\ \texttt{[]}\dots\texttt{[]}\ A_n\ \textbf{end;}\ A' \\
& \text{or } \textbf{case}\ E\ \textbf{is}\ P_0\texttt{->}A_0\ \texttt{|}\dots\texttt{|}\ P_n\texttt{->}A_n\ \textbf{end;}\ A' \\
\mathit{ucuns}(A_1\textbf{;}\ A_1\textbf{;}\ A_2) \wedge \mathit{ucuns}(A_2) & \text{if } A \text{ has the form } \textbf{while}\ E\ \textbf{do}\ A_1\ \textbf{end;}\ A_2 \\
\textbf{false} & \text{if } A \text{ has the form } G\ O_1 \dots O_n\ Q\ \textbf{in}\ W \\
\mathit{ucuns}(A\textbf{;}\ \textbf{null}) & \text{otherwise}
\end{cases}
$$

$$
\mathit{reach}(A) = \begin{cases}
\textbf{true} & \text{if } A \text{ has the form } \textbf{to}\ s',\ \textbf{stop},\ \textbf{to}\ s'\textbf{;}\ A_1, \text{ or } \textbf{stop;}\ A_1 \\
\textbf{false} & \text{if } A \text{ has the form } \textbf{null},\ G\ O_1 \dots O_n\ Q\ \textbf{in}\ W, \\
& G\ O_1 \dots O_n\ Q\ \textbf{in}\ W\textbf{;}\ A_1, \\
& \textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end},\ \textbf{while}\ E\ \textbf{do}\ A_0\ \textbf{end;}\ A_1,\ \textbf{wait}\ E, \\
& \text{or } \textbf{wait}\ E\textbf{;}\ A_1 \\
\mathit{reach}(A_1) & \text{if } A \text{ has the form } A_0\textbf{;}\ A_1 \text{ where } A_0 \text{ has the form} \\
& V_0\textbf{,}\dots\textbf{,}V_n\ \texttt{:=}\ E_0\textbf{,}\dots\textbf{,}E_n, \\
& V_0\textbf{,}\dots\textbf{,}V_n\ \texttt{:=}\ \textbf{any}\ T_0\textbf{,}\dots\textbf{,}T_n\ \textbf{where}\ E, \\
& \textbf{reset}\ V_0\textbf{,}\dots\textbf{,}V_n, \text{ or } \textbf{null} \\
\mathit{reach}(A_1\textbf{;}\ A_3) \wedge \mathit{reach}(A_2\textbf{;}\ A_3) & \text{if } A \text{ has the form } \textbf{if}\ E\ \textbf{then}\ A_1\ \textbf{else}\ A_2\ \textbf{end;}\ A_3 \\
\bigwedge_{i\in 0..n} \mathit{reach}(A_i\textbf{;}\ A') & \text{if } A \text{ has the form } \textbf{select}\ A_0\ \texttt{[]}\dots\texttt{[]}\ A_n\ \textbf{end;}\ A' \\
\mathit{exh}(\{\langle P_0\rangle,\dots,\langle P_n\rangle\}) & \\
\quad \wedge\ \bigwedge_{i\in 0..n} \mathit{reach}(A_i\textbf{;}\ A') & \text{if } A \text{ has the form } \textbf{case}\ E\ \textbf{is}\ P_0\texttt{->}A_0\ \texttt{|}\dots\texttt{|}\ P_n\texttt{->}A_n\ \textbf{end;}\ A' \\
\mathit{reach}(A\textbf{;}\ \textbf{null}) & \text{otherwise}
\end{cases}
$$

Figure A.1: Unicity of communication and undelayed next state reachability

A.1.2 Equivalent definition of validity-stable synchronizers

The definition of *validity_stable* given in Section 4.5.2 uses a quantification over the power set of $\mathbb{U}$, therefore an algorithm which directly translates this definition would be of exponential complexity. In this section we present an alternative definition, corresponding to an algorithm of polynomial complexity.

First we define for each set $\mathcal{U} \in \mathbb{U}$ its *shadow* i.e., the set of units to which the elements of $\mathcal{U}$ are related by $\succ$. Formally:

$$shadow(\mathcal{U}) \stackrel{\text{def}}{=} \{u \in \mathbb{U} \mid (\exists u' \in \mathcal{U})\ u' \succ u \text{ or } u \succ u'\}$$

This allows us to define the predicate *validity_stable'* on the set of synchronizers by:

$$\begin{array}{ll} validity\_stable'(G) \text{ iff} & valid\_active(start(G)) \wedge \\ & (\forall u \in start(G), \mathcal{U} \in sync(R))\ valid\_active(\mathcal{U}) \Rightarrow \\ & \quad ((\{u\} \cup shadow(\{u\})) \setminus shadow(\mathcal{U})) \subseteq stop(G) \end{array}$$

Proposition A.1. Let be assumed that all synchronizers G satisfy $(\exists \mathcal{U}_0 \in sync(G))$ $valid\_active(\mathcal{U}_0)$. Then the predicates *validity_stable* given in 4.5.2 and *validity_stable'* given here are equal.

During the proof, we use two lemmas, which are direct consequences from the definition of the predicate *valid_active*:

Lemma A.1. Let $\mathcal{U}_1, \mathcal{U}_2 \subseteq \mathbb{U}$ be sets.
$valid\_active(\mathcal{U}_1 \cup \mathcal{U}_2) \Rightarrow valid\_active(\mathcal{U}_1) \wedge valid\_active(\mathcal{U}_2)$

Lemma A.2. Let $\mathcal{U}_1, \mathcal{U}_2 \subseteq \mathbb{U}$ be sets.
$(valid\_active(\mathcal{U}_1) \text{ and } (\forall u \in \mathcal{U}_1)\ valid\_active(\mathcal{U}_2 \cup \{u\})) \Rightarrow valid\_active(\mathcal{U}_1 \cup \mathcal{U}_2)$

Proof. (of Proposition A.1)

We have to show for an arbitrary synchronizer G that

$validity\_stable(G)$ iff $validity\_stable'(G)$

"$\Rightarrow$": Let be $validity\_stable(G)$.

1. By assumption, there is a $\mathcal{U}_0 \in sync(G)$ such that $valid\_active(\mathcal{U}_0)$, and then by hypothesis we have
$valid\_active((\mathcal{U}_0 \setminus stop(G)) \cup start(G))$, thus by Lemma A.1 $valid\_active(start(G))$.

2. Let $u \in start(G), \mathcal{U} \in sync(G)$ be such that $valid\_active(\mathcal{U})$.

Let $u' \in ((\{u\} \cup shadow(\{u\})) \setminus shadow(\mathcal{U}))$. It has to be shown that $u' \in stop(G)$. To this aim, we distinguish three possible cases:

- Case $u' = u$: By $u' \notin shadow(\mathcal{U})$, we know that $valid\_active(\mathcal{U} \cup \{u'\})$. Also, by case hypothesis, $u' \in start(G)$. By definition of $validity\_stable(G)$, we then have $u' \notin ((\mathcal{U} \cup \{u'\}) \setminus stop(G))$, and thus $u' \in stop(G)$

- Case $u' \in shadow(\{u\})$ and $u' \in \mathcal{U}$: Thus we have $u \succ u'$ or $u' \succ u$, and then $\neg valid\_active(\{u', u\})$. Meanwhile, by condition and Lemma A.1, we know $validity\_active((\mathcal{U} \cup \{u\}) \setminus stop(G))$, and thus $validity\_active((\mathcal{U} \cup \{u, u'\}) \setminus stop(G))$. From this we conclude $u' \in stop(G)$.

- Case $u' \in shadow(\{u\})$ and $u' \notin \mathcal{U}$: As we have $u' \notin shadow(\mathcal{U})$, we can conclude $valid\_active(\mathcal{U} \cup \{u'\})$. Also, we always have $\neg valid\_active(\{u', u\})$, thus, by definition of $validity\_stable(G)$, we get $u' \in stop(G)$.

"$\Leftarrow$": Let be $validity\_stable'(G)$.

Let the set $\mathcal{U} \subseteq \mathbb{U}$ be such that $valid\_active(\mathcal{U})$ and the set $\mathcal{U}_s \in sync(G)$ be such that $\mathcal{U}_s \subseteq \mathcal{U}$. (By Lemma A.1, we notice that $valid\_active(\mathcal{U}_s)$.) Now two statements, given by the definition of *validity_stable*, then have to be shown:

First, we need to show that $(\mathcal{U} \setminus stop(G)) \cap start(G) = \varnothing$. Let us assume that there is $u \in ((\mathcal{U} \setminus stop(G)) \cap start(G))$.

As we must have $u \in \mathcal{U}$, we clearly have also $u \notin shadow(\mathcal{U}_s)$. By $u \in start(G)$, the definition of $validity\_stable'(G)$ states then that $u \in stop(G)$, which contradicts the assumption; it is thus impossible.

Second, we need to show that $valid\_active((\mathcal{U} \setminus stop(G)) \cup start(G))$.

As we already know that $valid\_active(start(G))$, Lemma A.2 tells us that it is sufficient to show for an arbitrary $u_0 \in start(G)$ that $valid\_active((\mathcal{U} \setminus stop(G)) \cup \{u_0\})$.

Thus, we have to show that, if there is a $u_1 \in \mathcal{U}$ such that $u_1 \succ u_0$ or $u_0 \succ u_1$, we then also have $u_1 \in stop(G)$. The hypothesis "$u_1 \succ u_0$ or $u_0 \succ u_1$" is equivalent to "$u_1 \in shadow(\{u_0\})$".

Case $u_1 \in \mathcal{U}_s$: By definition of *shadow*, we have $u_1 \notin shadow(\mathcal{U}_s)$, and thus by the second conclusion of the definition of *validity_stable'* we deduce $u_1 \in stop(G)$.

Case $u_1 \notin \mathcal{U}_s$: By $valid\_active(\mathcal{U})$ we see that $valid\_active(\mathcal{U}_s \cup \{u_1\})$, and thus that $u_1 \notin shadow(\mathcal{U}_s)$, thus (as above) $u_1 \in stop(G)$. □

Remark A.1. *The assumption used in Proposition A.1 does not impose a real restriction, as a synchronizer not satisfying it could obviously not be used in a discrete transition, thus the dynamic semantics of a module is equal for the two predicates used in a dynamics semantics restriction. The proposition does not hold without the restriction, as the following example shows:*

*Let "***sync** *G* **is** *u* **and** *u'* **start** *u, u'* **end sync***" be a synchronizer with u, u' units such that $u \succ u'$. Then $validity\_stable(G)$, but not $validity\_stable'(G)$.*

Example A.1. *We illustrate the defintion of validity_stable′ by recalling the synchronizer Finish_Braking of Example 4.3 on page 78. Here, we suppose that our unit Brakes is a subunit (possibly among others) of a unit named Car_Control, which itself is not a subunit.*

To verify if validity_stable′(Finish_Braking) is satisfied, we begin by checking the predicate valid_active(start(Finish_Braking)). As start(Finish_Braking) is the singleton {Brakes}, this is clearly true.

Next, we have to check the condition that quantifies over all $u \in \mathit{start}(\mathit{Finish\_Braking})$ *and* $\mathcal{U} \in \mathit{sync}(\mathit{Finish\_Braking})$. *This is simple, as there is only one element each i.e.,* $u = \mathit{Brakes}$ *and* $\mathcal{U} = \{\mathit{Front\_Brakes}, \mathit{Rear\_Brakes}\}$. *The set* $\mathcal{U}$ *is valid_active, therefore we have to check if the following is satisfied:*

$$((\{\mathit{Brakes}\} \cup \mathit{shadow}(\{\mathit{Brakes}\})) \setminus \mathit{shadow}(\{\mathit{Front\_Brakes}, \mathit{Rear\_Brakes}\})) \subseteq \{\mathit{Front\_Brakes}, \mathit{Rear\_Brakes}\}$$

By the definition above, we have

$$\mathit{shadow}(\{\mathit{Brakes}\}) = \{\mathit{Car\_Control}, \mathit{Front\_Brakes}, \mathit{Rear\_Brakes}\}$$

and

$$\mathit{shadow}(\{\mathit{Front\_Brakes}, \mathit{Rear\_Brakes}\}) = \{\mathit{Car\_Control}, \mathit{Brakes}\}.$$

Thus, the inequation simplifies to

$$(\{\mathit{Car\_Control}, \mathit{Brakes}, \mathit{Front\_Brakes}, \mathit{Rear\_Brakes}\} \setminus \{\mathit{Car\_Control}, \mathit{Brakes}\}) \subseteq \{\mathit{Front\_Brakes}, \mathit{Rear\_Brakes}\}$$

which is clearly true.

A.1.3 Variable initialization

The following algorithm, already briefly described in Section 4.6.2, verifies for a given ATLANTIF module if each variable is assigned a value before being read. It is composed of three steps as follows:

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 with 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. A.2. 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'[34]. 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.

Intuitively, the first two of these sets extend to execution paths the sets *use* and *def* we defined in Section 4.3 for expressions, patterns, and offers.

[34]With the exception of paths containing a **while** loop, which are not split up into different paths, given that there are in general infinitely many.

$$
\begin{aligned}
&\mathit{local\_edges}(\mathbf{null}) \stackrel{\text{def}}{=} \{(\varnothing, \varnothing, \varnothing, \varepsilon, \delta)\} \\
&\mathit{local\_edges}(\mathbf{wait}\ E) \stackrel{\text{def}}{=} \{(\mathit{use}(E), \varnothing, \varnothing, \varepsilon, \delta)\} \\
&\mathit{local\_edges}(V_0, \ldots, V_n\ \texttt{:=}\ E_0, \ldots, E_n) \stackrel{\text{def}}{=} \{(\textstyle\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\ \texttt{:=}\ \mathbf{any}\ T_0, \ldots, T_n\ \mathbf{where}\ E) \stackrel{\text{def}}{=} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \{(\mathit{use}(E), \{V_0, \ldots, V_n\}, \varnothing, \varepsilon, \delta)\} \\
&\mathit{local\_edges}(\mathbf{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\ \mathbf{in}\ W) \stackrel{\text{def}}{=} \\
&\qquad \{(\textstyle\bigcup_{i \in 1..n} (\mathit{use}(O_i) \setminus (\bigcup_{j \in 1..(i-1)} \mathit{def}(O_j))) \cup \mathit{read}(W), \bigcup_{i \in 1..n} \mathit{def}(O_i), \varnothing, G, \delta)\} \\
&\mathit{local\_edges}(\mathbf{to}\ s') \stackrel{\text{def}}{=} \{(\varnothing, \varnothing, \varnothing, \varepsilon, s')\} \\
&\mathit{local\_edges}(\mathbf{select}\ A_0\, \texttt{[]} \ldots \texttt{[]}\, A_n\ \mathbf{end}) \stackrel{\text{def}}{=} \textstyle\bigcup_{i \in 0..n} \mathit{local\_edges}(A_i) \\
&\mathit{local\_edges}(\mathbf{case}\ E\ \mathbf{is}\ P_0 \texttt{->} A_0 \,\texttt{|} \ldots \texttt{|}\, P_n \texttt{->} A_n\ \mathbf{end}) \stackrel{\text{def}}{=} \\
&\quad \textstyle\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\qquad (\mathcal{V}_o, \mathcal{V}_a, \mathcal{V}_r, l, s') \in \mathit{local\_edges}(A_i)\} \\
&\mathit{local\_edges}(\mathbf{if}\ E\ \mathbf{then}\ A_1\ \mathbf{else}\ A_2\ \mathbf{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}(\mathbf{while}\ E\ \mathbf{do}\ A_0\ \mathbf{end}) \stackrel{\text{def}}{=} \{(\mathit{use}(E) \cup \textstyle\bigcup_{i \in 0..n} \mathcal{V}_o^i, \varnothing, \bigcup_{i \in 0..n} \mathcal{V}_r^i, \varepsilon, \delta)\} \\
&\qquad\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\, \texttt{;}\, 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\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\quad \text{raise error, if } \mathcal{V}_r \cap \mathcal{V}'_o \neq \varnothing
\end{aligned}
$$

Figure A.2: Function *local_edges*

Note that the function *local_edges* already notices a trivial case of variable initialization errors (and raises an exception), which is the case of a sequential composition $A_1 ; A_2$ such that A_1 resets a variable that is read in A_2.

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 $sync(G)$. During the latter case, units may be stopped and/or started, according to G.

The algorithm given in Fig. A.3 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 4.6.3 on page 81. It uses the predicate *next_*π from Section 4.6.3.

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. A.4, 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 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.

The module is well-initialized, if the algorithm terminates without raising an error and *set_before*(n_0) is a subset of $dom(\rho_0)$.

Proposition A.2. The algorithm defined in this section is sufficient to detect in an ATLANTIF module any variable that may be used without being defined, but is not necessary.

Proof. Sufficient: Suppose M is a module allowing a run in which a variable V is used without being assigned a value. Three different reasons are possible:

- There is a unit and a multibranch transition allowing an execution path where V is reset before being used.
- V has neither an initial value nor any assignement was made to V before it should be used.
- V has an initial value and/or an assignement was made to V before it should be used, but it has been reset since by a **reset** action or by a unit desactivation.

In the first case, the function *local_edges* will raise an error during the construction of the local variable usage graph (last case of the definition in Fig. A.2).

$VUG \leftarrow \varnothing$
$Tmp \leftarrow \{(\pi_0)\}$
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 A.3: Pseudocode for variable usage graph construction

for each $n \in VUG$
  $set\_before(n) \leftarrow \mathcal{V}$
  $explored(n) \leftarrow$ **false**
end for
$set\_before(n_0) \leftarrow \varnothing$
while $((\exists n \in VUG) explored(n) =$ **false**$)$
  choose $n \in VUG$ with $explored(n) =$ **false**
    $explored(n) =$ **true**
    for each $(\mathcal{V}_o, \mathcal{V}_a, \mathcal{V}_r, \pi') \in \mathcal{K}$ (where $n = (\pi, \mathcal{K})$)
      if $((\exists V \in \mathcal{V}_o)\ V \notin set\_before(n)$ then raise error end if
      find $n' = (\pi', \mathcal{K}') \in VUG$ (for the π' given above)
      $Tmp \leftarrow set\_before(n') \cap ((set\_before(n) \setminus \mathcal{V}_r) \cup \mathcal{V}_a)$
      if $((Tmp \neq set\_before(n'))$ or $(explored(n') =$ **false**$))$ then
        $set\_before(n') \leftarrow Tmp$
        $explored(n') \leftarrow$ **false**
      end if
    end for
  end choose
end while

Figure A.4: Pseudocode for fix-points of variable definitions in the variable usage graph

In the two other cases, the run has a corresponding path in the global variable usage graph (VUG). In the second case, this path starts from the initial node n_0 of VUG. It goes by the nodes $n_1, \ldots, n_k$, where n_k corresponds to the state in the run that directly preceeds V being used. By the definition of Fig. A.4, we then have $set\_before(n_k)$, thus $set\_before(n_{k-1})$, ..., thus $set\_before(n_0)$. Therefore, M is not well-initialized, because $set\_before(n_0) \not\subseteq dom(\rho_0)$.

In the third case, we can apply the same argumentation as in the second case, except for ignoring the beginning of the path in VUG until n_l is reached, which is the node following directly the last reset of V.

Not necessary: The following counterexample shows that the algorithm is not necessary. We suppose a module with a single unit and the according synchronizers, the unit beginning with the following multibranch transitions:

from s_1
  reset V_1; V_2 := **true**; **to** s_3
from s_2
  V_1, V_2 := 5, **false**; **to** s_3
from s_3
  if (**not** V_2) **then** V_3 := V_1 **end**; **to** s_4

Then our algorithm will require the variable V_1 to have a value assigned each time the state s_3 is reached, which is not satisfied by the execution path of s_1, therefore an error will be raised. However, because of the assignment made to V_2 following s_1, the execution path of s_3 that reads V_1 could not be taken, thus it cannot happen in the example that the variable V_1 is used without being assigned a value.

Note that false errors raised by our algorithm can not only be caused by non-considering of data constraints, but also by non-considering of timing constraints or by considering invalid chains of **silent** synchronizers. These imprecisions are due to abstractions we made in the definition of the global variable usage graph in order to limit its size. ☐

Differences to NTIF. The static semantics definition of NTIF also requires all variables to be well-initialized and applies an algorithm (cf. Appendix A.3 of [61]) sharing important ideas. However, changes were made regarding the scope of the algorithm and the efficiency.

Regarding the scope, the new algorithm consideres constructs that do not exist in NTIF, namely a limited variable sharing and unit starting and stopping (also delay actions and time windows, but those are, for the concerns of this section, trivial extensions). The possibility of sharing variables makes it necessary for our algorithm to consider the module as a whole (instead of verifying one by one the units), and the starting and stopping represents a new source of variable assignments and variable resets.

Regarding the efficiency, the NTIF-algorithm applied the evaluation of multibranch transitions (finding execution paths, finding assigned, read, and reset variables) in each step of the minimal fix-point search; where the number of those steps may be exponential depending on the number of discrete states. Our algorithm applies the same fix-point search (which is thus also of exponential complexity), but it does not contain the evaluation of multibranch transitions, which is done during the construction of the variable usage graph i.e., each multibranch transition is only evaluated once.

A.2 Translation to graphical models

A.2.1 Translation of an ATLANTIF unit to a TINA TPN

In this section, we will provide more detailed definitions for the translation given in Section 6.2.3. There, we translate ATLANTIF units to prelaminary TPN by two main steps, the first of which is to evaluate all execution paths of the unit (mainly defined by the function *trans_p*, given in Fig. 6.10 on page 156). The second step is to create auxiliary construct that represent the timing constraints, including the "emptying" i.e., the introduction of more auxiliary constructs that delete tokens from auxiliary places. The algorithm of the second step is given by the pseudocode in Fig. A.5 to A.12.

In the pseudocode, each place is represented by a triple, build from a unique identifier, a list of (the identifiers of) auxiliary places, and a label (e.g., the identifier of the corresponding ATLANTIF discrete state). Depending on the context, we identify a place either such a triple (id, AP, p) or simply by p or id. Each transition is represented by a 6-tuple, build from a unique identifier, a list of in-places, a list of inhibitor-places, a list of out-places, a label (a gate or ε), and a static firing interval.

The generation of identifiers for places and transitions used in the algorithm is understood to produce *unique* identifiers. In the translator implementation of the atlantif tool

given a set of places P'_u, a set of transitions T'_u, and a set of execution path evaluations $\mathcal{T}_u$
$P \leftarrow \varnothing, T \leftarrow \varnothing, Pr \leftarrow \varnothing$
for all $(id, \varnothing, p) \in P'_u$
  $T_{new} \leftarrow \varnothing, \tilde{P}_{aux} \leftarrow \varnothing$
  find $T_p \subseteq T'_u$ by $T_p = \{t \in T_p \mid in(t) = \{p\}\}$ *(note that all in-sets are singletons)*
  for all $t \in T_p$
    find in $\mathcal{T}_u$ the $\Pi = (I, O, l, w, x, E, A)$ according to t (i.e., $t_\Pi = t$)
    in case the tuple (w, x) (i.e., interval and modality) is the following:
      $([0, k], \textbf{false}) \ \rightarrow$
        check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon,]k, \infty[)$
          yes $\rightarrow$
            modify Π : add id'_1 as in-place and set w to $[0, \infty[$
            add priority $id'_2 > id$ to Pr
          no $\rightarrow$
            generate new place identifier id_1 and new transition identifier id_2
            $P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
            $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
            $T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon,]k, \infty[)\}$
            modify Π : add id_1 as in-place and set w to $[0, \infty[$
            add priority $id_2 > id$ to Pr
            $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
      $([0, k], \textbf{true}) \ \rightarrow$
        check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon, [k, k])$
          yes $\rightarrow$
            modify Π : add id'_1 as in-place and set w to $[0, \infty[$
          no $\rightarrow$
            generate new place identifier id_1 and new transition identifier id_2
            $P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
            $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
            $T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon, [k, k])\}$
            modify Π : add id_1 as in-place and set w to $[0, \infty[$
            add priority $id > id_2$ to Pr
            $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$

(continued in Fig. A.6)

Figure A.5: Pseudocode for the second step of the translation from a unit to a TPN (1)

$([m,k],\mathbf{false}) \ \rightarrow$
  check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon, [m,\infty[)$
    yes $\rightarrow$
      check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon,]k,\infty[)$
        yes $\rightarrow$
          modify Π : add id'_3 as in-place, id'_1 as inhibitor-place, and set w to $[0,\infty[$
          add priority $id'_4 > id$ to Pr
        no $\rightarrow$
          generate new place identifier id_3 and new transition identifier id_4
          $P \leftarrow P \cup \{(id_3, \varnothing, p\_aux)\}$
          $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_3\}$
          $T_{new} \leftarrow T_{new} \cup \{(id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon,]k,\infty[)\}$
          modify Π : add id_3 as in-place, id'_1 as inhibitor-place, and set w to $[0,\infty[$
          add priority $id_4 > id$ to Pr
          $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_4\}$
    no $\rightarrow$
      check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon,]k,\infty[)$
        yes $\rightarrow$
          generate new place identifier id_1 and new transition identifier id_2
          $P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
          $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
          $T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon, [m,\infty[)\}$
          modify Π : add id'_3 as in-place, id_1 as inhibitor-place, and set w to $[0,\infty[$
          add priority $id'_4 > id$ to Pr
          $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
        no $\rightarrow$
          generate new place identifiers id_1, id_3 and new transition identifiers id_2, id_4
          $P \leftarrow P \cup \{(id_1, \varnothing, p\_aux), (id_3, \varnothing, p\_aux)\}$
          $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1, id_3\}$
          $T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon, [m,\infty[), (id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon,]k,\infty[)\}$
          modify Π : add id_3 as in-place, id_1 as inhibitor-place, and set w to $[0,\infty[$
          add priority $id_4 > id$ to Pr
          $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2, id_4\}$

(continued in Fig. A.7)

Figure A.6: Pseudocode for the second step of the translation from a unit to a TPN (2)

$([m, k], \mathbf{true}) \;\rightarrow$
check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon, [m, \infty[)$
yes $\rightarrow$
check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon, [k, k])$
yes $\rightarrow$
modify Π : add id'_3 as in-place, id'_1 as inhibitor-place, and set w to $[0, \infty[$
add priorities $id > id'_4, id'_2 > id'_4$ to Pr
no $\rightarrow$
generate new place identifier id_3 and new transition identifier id_4
$P \leftarrow P \cup \{(id_3, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_3\}$
$T_{new} \leftarrow T_{new} \cup \{(id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon, [k, k])\}$
modify Π : add id_3 as in-place, id'_1 as inhibitor-place, and set w to $[0, \infty[$
add priorities $id > id_4, id'_2 > id_4$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_4\}$
no $\rightarrow$
check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon, [k, k])$
yes $\rightarrow$
generate new place identifier id_1 and new transition identifier id_2
$P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
$T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon, [m, \infty[)\}$
modify Π : add id'_3 as in-place, id_1 as inhibitor-place, and set w to $[0, \infty[$
add priorities $id > id'_4, id_2 > id'_4$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
no $\rightarrow$
generate new place identifiers id_1, id_3 and new transition identifiers id_2, id_4
$P \leftarrow P \cup \{(id_1, \varnothing, p\_aux), (id_3, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1, id_3\}$
$T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon, [m, \infty[), (id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon, [k, k])\}$
modify Π : add id_3 as in-place, id_1 as inhibitor-place, and set w to $[0, \infty[$
add priorities $id > id_4, id_2 > id_4$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2, id_4\}$
$([0, k[, \mathbf{any\ bool}) \;\rightarrow$ *(note that by* ATLANTIF *static semantics* $x =$ **false***)*
check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)$
yes $\rightarrow$
modify Π : add id'_1 as in-place and set w to $[0, \infty[$
add priority $id'_2 > id$ to Pr
no $\rightarrow$
generate new place identifier id_1 and new transition identifier id_2
$P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
$T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)\}$
modify Π : add id_1 as in-place and set w to $[0, \infty[$
add priority $id_2 > id$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
(continued in Fig. A.8)

Figure A.7: Pseudocode for the second step of the translation from a unit to a TPN (3)

$([m,k[$, **any bool**$)$ $\rightarrow$ *(note that by* ATLANTIF *static semantics* $x =$ **false***)*
check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon, [m, \infty[)$
yes $\rightarrow$
check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)$
yes $\rightarrow$
modify Π : add id'_3 as in-place, id'_1 as inhibitor-place, and set w to $[0, \infty[$
add priority $id'_4 > id$ to Pr
no $\rightarrow$
generate new place identifier id_3 and new transition identifier id_4
$P \leftarrow P \cup \{(id_3, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_3\}$
$T_{new} \leftarrow T_{new} \cup \{(id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)\}$
modify Π : add id_3 as in-place, id'_1 as inhibitor-place, and set w to $[0, \infty[$
add priority $id_4 > id$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_4\}$
no $\rightarrow$
check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)$
yes $\rightarrow$
generate new place identifier id_1 and new transition identifier id_2
$P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
$T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon, [m, \infty[)\}$
modify Π : add id'_3 as in-place, id_1 as inhibitor-place, and set w to $[0, \infty[$
add priority $id'_4 > id$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
no $\rightarrow$
generate new place identifiers id_1, id_3 and new transition identifiers id_2, id_4
$P \leftarrow P \cup \{(id_1, \varnothing, p\_aux), (id_3, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1, id_3\}$
$T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon, [m, \infty[), (id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)\}$
modify Π : add id_3 as in-place, id_1 as inhibitor-place, and set w to $[0, \infty[$
add priority $id_4 > id$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2, id_4\}$
$([0,\infty[$, **any bool**$)$ $\rightarrow$
do nothing
$([m,\infty[$, **any bool**$)$ $\rightarrow$
check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon, [m, \infty[)$
no $\rightarrow$
modify Π : add id'_1as inhibitor-place and set w to $[0, \infty[$
no $\rightarrow$
generate new place identifier id_1 and new transition identifier id_2
$P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
$T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon, [m, \infty[)\}$
modify Π : add id_1as inhibitor-place and set w to $[0, \infty[$
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
(continued in Fig. A.9)

Figure A.8: Pseudocode for the second step of the translation from a unit to a TPN (4)

$(]m, k], \mathbf{false}) \ \rightarrow$
  check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[)$
    yes $\rightarrow$
      check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon,]k, \infty[)$
        yes $\rightarrow$
          modify Π : add id'_3 as in-place, id'_1 as inhibitor-place, and set w to $[0, \infty[$
          add priority $id'_4 > id$ to Pr
        no $\rightarrow$
          generate new place identifier id_3 and new transition identifier id_4
          $P \leftarrow P \cup \{(id_3, \varnothing, p\_aux)\}$
          $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_3\}$
          $T_{new} \leftarrow T_{new} \cup \{(id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon,]k, \infty[)\}$
          modify Π : add id_3 as in-place, id'_1 as inhibitor-place, and set w to $[0, \infty[$
          add priority $id_4 > id$ to Pr
          $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_4\}$
    no $\rightarrow$
      check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon,]k, \infty[)$
        yes $\rightarrow$
          generate new place identifier id_1 and new transition identifier id_2
          $P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
          $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
          $T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[)\}$
          modify Π : add id'_3 as in-place, id_1 as inhibitor-place, and set w to $[0, \infty[$
          add priority $id'_4 > id$ to Pr
          $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
        no $\rightarrow$
          generate new place identifiers id_1, id_3 and new transition identifiers id_2, id_4
          $P \leftarrow P \cup \{(id_1, \varnothing, p\_aux), (id_3, \varnothing, p\_aux)\}$
          $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1, id_3\}$
          $T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[), (id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon,]k, \infty[)\}$
          modify Π : add id_3 as in-place, id_1 as inhibitor-place, and set w to $[0, \infty[$
          add priority $id_4 > id$ to Pr
          $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2, id_4\}$

(continued in Fig. A.10)

Figure A.9: Pseudocode for the second step of the translation from a unit to a TPN (5)

$(]m, k], \mathbf{true}) \;\rightarrow$
  check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[)$
    yes $\rightarrow$
      check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon, [k, k])$
        yes $\rightarrow$
          modify Π : add id'_3 as in-place, id'_1 as inhibitor-place, and set w to $[0, \infty[$
          add priorities $id > id'_4, id'_2 > id'_4$ to Pr
        no $\rightarrow$
          generate new place identifier id_3 and new transition identifier id_4
          $P \leftarrow P \cup \{(id_3, \varnothing, p\_aux)\}$
          $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_3\}$
          $T_{new} \leftarrow T_{new} \cup \{(id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon, k, k])\}$
          modify Π : add id_3 as in-place, id'_1 as inhibitor-place, and set w to $[0, \infty[$
          add priorities $id > id_4, id'_2 > id_4$ to Pr
          $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_4\}$
    no $\rightarrow$
      check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon, [k, k])$
        yes $\rightarrow$
          generate new place identifier id_1 and new transition identifier id_2
          $P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
          $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
          $T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[)\}$
          modify Π : add id'_3 as in-place, id_1 as inhibitor-place, and set w to $[0, \infty[$
          add priorities $id > id'_4, id_2 > id'_4$ to Pr
          $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
        no $\rightarrow$
          generate new place identifiers id_1, id_3 and new transition identifiers id_2, id_4
          $P \leftarrow P \cup \{(id_1, \varnothing, p\_aux), (id_3, \varnothing, p\_aux)\}$
          $\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1, id_3\}$
          $T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[), (id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon, [k, k])\}$
          modify Π : add id_3 as in-place, id_1 as inhibitor-place, and set w to $[0, \infty[$
          add priorities $id > id_4, id_2 > id_4$ to Pr
          $\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2, id_4\}$

(continued in Fig. A.11)

Figure A.10: Pseudocode for the second step of the translation from a unit to a TPN (6)

($]m,k[$, **any bool**) $\rightarrow$ *(note that by* ATLANTIF *static semantics* $x =$ **false***)*
check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[)$
yes $\rightarrow$
check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)$
yes $\rightarrow$
modify Π : add id'_3 as in-place, id'_1as inhibitor-place, and set w to $[0, \infty[$
add priority $id'_4 > id$ to Pr
no $\rightarrow$
generate new place identifier id_3 and new transition identifier id_4
$P \leftarrow P \cup \{(id_3, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_3\}$
$T_{new} \leftarrow T_{new} \cup \{(id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)\}$
modify Π : add id_3 as in-place, id'_1as inhibitor-place, and set w to $[0, \infty[$
add priority $id_4 > id$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_4\}$
no $\rightarrow$
check if there is a $t'' \in T_{new}$ such that $t'' = (id'_4, \{id'_3\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)$
yes $\rightarrow$
generate new place identifier id_1 and new transition identifier id_2
$P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
$T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[)\}$
modify Π : add id'_3 as in-place, id_1as inhibitor-place, and set w to $[0, \infty[$
add priority $id'_4 > id$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
no $\rightarrow$
generate new place identifiers id_1, id_3 and new transition identifiers id_2, id_4
$P \leftarrow P \cup \{(id_1, \varnothing, p\_aux), (id_3, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1, id_3\}$
$T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[), (id_4, \{id_3\}, \varnothing, \varnothing, \varepsilon, [k, \infty[)\}$
modify Π : add id_3 as in-place, id_1as inhibitor-place, and set w to $[0, \infty[$
add priority $id_4 > id$ to Pr
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2, id_4\}$
($]m, \infty[$, **any bool**) $\rightarrow$
check if there is a $t' \in T_{new}$ such that $t' = (id'_2, \{id'_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[)$
no $\rightarrow$
modify Π : add id'_1as inhibitor-place and set w to $[0, \infty[$
no $\rightarrow$
generate new place identifier id_1 and new transition identifier id_2
$P \leftarrow P \cup \{(id_1, \varnothing, p\_aux)\}$
$\tilde{P}_{aux} \leftarrow \tilde{P}_{aux} \cup \{id_1\}$
$T_{new} \leftarrow T_{new} \cup \{(id_2, \{id_1\}, \varnothing, \varnothing, \varepsilon,]m, \infty[)\}$
modify Π : add id_1as inhibitor-place and set w to $[0, \infty[$
$\mathcal{T}_\beta \leftarrow \mathcal{T}_\beta \cup \{id_2\}$
end for *(loop on execution paths from p)*
(continued in Fig. A.12)

Figure A.11: Pseudocode for the second step of the translation from a unit to a TPN (7)

```
if $(id, \varnothing, p)$ represents the initial discrete state of $u$ then the
      initial marking of $\mathcal{R}_u$ is $\{id\} \cup \tilde{P}_{aux}$
$P \leftarrow P \cup \{(id, \tilde{P}_{aux}, p)\}$
if $(\tilde{P}_{aux} \neq \varnothing)$ then      (begin emptying construction)
  for each labelled transition $t$ with $p$ as in-place
    find in $\mathcal{T}_u$ the $\Pi = (I, O, l, w, x, E, A)$ according to $t$ (i.e., $t_\Pi = t$)
    $\tilde{P}_{toempty} \leftarrow \tilde{P}_{aux} \setminus \{id' \mid id'$ denotes the in-place of an auxiliary transition of $t\}$
    (we suppose this set $\tilde{P}_{toempty}$ to be ordered)
    $m \leftarrow card(\tilde{P}_{toempty})$
    if $(m > 0)$ then      (i.e., are there actually places to empty?)
      $k \leftarrow 2^m - 1; j \leftarrow 0$
      generate new place identifier $id_1$
      $P \leftarrow P \cup \{(id_1, \varnothing, p\_out)\}$
      loop on $j$ by increasing it by 1 each step, until $(j = k)$ included
        $new\_in \leftarrow id_1; l \leftarrow m; j' \leftarrow j$
        loop on $l$ by decreasing it by 1 each step, until $(l = 1)$ included
          if $(j' \geq 2^{(l-1)})$ then
            $new\_in \leftarrow new\_in \cup \{$(the $l^{\text{th}}$ element of $\tilde{P}_{toempty}$)$\}$
            $j' \leftarrow j' - 2^{(l-1)}$                end if
        end loop on $l$
        generate new transition identifier $id_2$
        $T_{new} \leftarrow T_{new} \cup \{(id_2, new\_in, \varnothing, out(t), \varepsilon, [0, 0])\}$
        $\mathcal{T}_\gamma \leftarrow \mathcal{T}_\gamma \cup \{id_2\}$
        if $(j < k)$ then add priority $successor(id_2) > id_2$ to $Pr$ end if
      end loop on $j$
      modify $\Pi$ : set $id_1$as the only out-place
    end if
  end for
end if      (end emptying construction)
$T \leftarrow T \cup T_{new}$
update in $T$ all transition containing $id$ as outplace by adding $\tilde{P}_{aux}$ to its out-places
end for (loop on places in $P'_u$)
```

Figure A.12: Pseudocode for the second step of the translation from a unit to a TPN (8)

(cf. Section 6.4), we produce them by successive numerotations $p1, p2, \ldots$ and $t1, t2, \ldots$ respectively.

Note that the pseudocode we show here also includes the optimisation as defined on page 162, which enables the multiple using of auxiliary transitions. This is implemented by the code "check if there is t'/t'' ...".

The transition families $\mathcal{T}_\alpha, \mathcal{T}_\beta, \mathcal{T}_\gamma$ are defined as in Section 6.2.4, to better illustrate which priorities will be introduced in the composition of the different unit-TPNs (cf. Fig. 6.23 on page 167).

A.3 Translation to Fiacre

This section presents the prototype translator from Atlantif to the Fiacre model, which we already motivated and outlined in Section 6.3.

A.3.1 The Fiacre model

In this section, we give an overview of the syntax of Fiacre as defined in [15]. A Fiacre *program* consists of declarations of *types*, *constants*, *channels*, *processes*, and *components*, and terminates by an identifier of one of the declared processes and components. This last identifier denotes the main process (or component).

Type declarations follow the same ideas as in Atlantif; they will not be detailed here. Constant declarations will not be detailed either, because they do not create additional expressive power.

A Fiacre *port* corresponds to a *gate* in Atlantif and most other formats. Each port is associated with a *channel* declaration, which describes cardinality and types of the offers that are exchanged via this port.

Processes. A process declaration in Fiacre consists of a header and a body. The process header contains the synchronization interfaces of the process (i.e., a list of ports with applying channels) and the data parameters (i.e., a list of typed variables). The process body consists of the following:

- a list of discrete state identifiers
- an optional declaration of local variables (where each declaration may optionally contain an initial value)
- an optional *statement* (Fiacre statements correspond to Atlantif actions) to be executed during the initialization of the process
- a list of multibranch transitions

The multibranch transitions of FIACRE are very similar to those of ATLANTIF (unsurprisingly, because both models inherited them from NTIF): The keyword **from** is followed by a discrete state identifier (which has to be declared in the discrete state list above) and then by a statement. FIACRE statements are defined by the grammar given in Table A.1, where the occurring ports must be among those appearing in the header of the process.

statement ::=	
null	(*inaction*)
| P_0 `,` ... `,` P_n `:=` E_0 `,` ... `,` E_n	(*deterministic assignment*)
| V_0 `,` ... `,` V_n `:=` **any** [**where** E]	(*nondet. assignment*)
| **while** E **do** statement$_0$ **end** [**while**]	(*while-loop*)
| **foreach** V **do** statement$_0$ **end** [**foreach**]	(*loop over values*)
| **if** E_0 **then** statement$_0$ [**elsif** E_1 **then** statement$_1$	
... **elsif** E_n **then** statement$_n$] **else** statement$_{n+1}$ **end** [**if**]	(*conditional*)
| **select** statement$_0$ `[]` ... `[]` statement$_n$ **end** [**select**]	(*nondeterministic choice*)
| **case** E **of** P_0 `->` statement$_0$ `|` ... `|` P_n `->` statement$_n$ **end** [**case**]	(*deterministic choice*)
| **to** s	(*jump to discrete state*)
| statement$_1$ `;` statement$_2$	(*sequential composition*)
| port [`?`P_0 `,` ... `,` P_n [**where** E] | `!`E_0 `,` ... `,` E_n]	(*port communication*)

Table A.1: Syntax of FIACRE statements

Note that there are several small differences between ATLANTIF actions (cf. Table 4.3 on page 53) and FIACRE statements: Unlike ATLANTIF, FIACRE does not have **wait**, **reset**, or **stop** actions, nor does it provide communication actions with time constraints or mixed (emissions and receptions) offers. Unlike FIACRE, ATLANTIF does not have an "**elsif**" construct (cf. the discussion on possible extensions of ATLANTIF in Section 4.7.3 on page 103), assignments with patterns on the left, or a **foreach** loop.

As in NTIF and ATLANTIF, the first discrete state in the list of multibranch transitions is the initial discrete state of the process.

Components. Similar to a process declaration, a component declaration in FIACRE consists of a header and a body. The header is defined exactly as for processes; the body contains the following elements, each of which is optional:

- A declaration of local variables (as in processes).
- A declaration of local ports. Unlike the ports listed in the header, the elements of this list may each optionally also be defined with a time interval (not detailed here, as we only consider untimed FIACRE in this translator).
- A partial order on the set of the ports occurring in the header and in the local declarations.
- An initial statement (as in processes).

- A parallel composition of processes and components, defined according to the following grammar:

$$
\begin{array}{l}
\text{composition} ::= \\
\quad \textbf{par}\ [(\texttt{*}\ \mid\ \text{port}_1 \texttt{|} \ldots \texttt{|}\text{port}_n)\ \textbf{in}] \\
\quad\quad \left[(\texttt{*}\ \mid\ \text{port}^1_1\texttt{,} \ldots \texttt{,}\text{port}^1_{m_1})\ \texttt{->}\right]\ \text{compblock}_1 \\
\quad\quad\ \texttt{||}\ \ldots\ \texttt{||} \\
\quad\quad \left[(\texttt{*}\ \mid\ \text{port}^k_1\texttt{,} \ldots \texttt{,}\text{port}^k_{m_k})\ \texttt{->}\right]\ \text{compblock}_k \\
\quad \textbf{end}\ [\textbf{par}]
\end{array}
$$

  A "compblock" is either another composition or an instantiation of a process or a component. It can be seen that this parallel composition is similar to synchronization vectors such as those described in Section 5.5.1.

  The composition operator describes the parallel execution of k structures (processes, components, and compositions) and the possible synchronizations among them, using a slight variation of the generalized parallel composition operator defined in Fig. 5.7 on page 116, without the "among" construct "$\#m$" i.e., a synchronization on one of the ports $\text{port}_1, \ldots, \text{port}_n$ can only occur among all those structures $(\text{compblock}_1, \ldots, \text{compblock}_k)$ at the same time.

A.3.2 Problems to overcome

The following list contains the most important aspects of untimed Atlantif for which no direct translation to Fiacre exists. Instead, emulations had to be implemented. These problems concern above all aspects related to the Atlantif synchronizers, which in Fiacre have to be expressed using synchronization vectors.

- The stopping and starting of units has no corresponding concept in Fiacre. Therefore, we will emulate this in a similar way as in the Uppaal translator i.e., by introducing additional discrete states that represent inaction of a process.

- In particular, asynchronous termination does not exist in Fiacre: The only way to transform a process into the discrete state representing inaction is a synchronization in which the process to be stopped participates. Therefore, asynchronous termination has to be emulated by a synchronization (as we also did in the translator to Uppaal).

- The Fiacre synchronization vectors provide only a single combination of processes that can synchronize on a given port. Therefore, to translate the different synchronization sets of Atlantif we will define one port for each synchronization set. Subsequently, a renaming situated on a higher hierarchical level will regroup those different ports again into the according gate names.

A.3.3 Restrictions

For the translator presented in this section, the following restrictions apply (in addition to the limitation to untimed Atlantif mentioned above):

- Variables occurring in units other than the one where they are declared i.e., the usage of the limited variable sharing that Atlantif provides, are not supported. In Fiacre, read/write-conflicts and write/write-conflicts on shared variables are excluded by a very restrictive usage of such variables within one multibranch transition, whereas in Atlantif, variables that can be accessed in different units can be used in transitions like any other variables, but restrictions exist on which units may read and/or write these variables (cf. Section 4.4.4). It seems possible to overcome these differences in a translator to Fiacre, but it is clearly not a trivial problem.

- Type declarations and function declarations being not yet formalized in Atlantif, they are not covered by this translator.

- Offers in communication actions must be of type integer. This restriction is due to the fact that Fiacre ports are typed, while Atlantif gates are not typed. Thus, a translator would have to contain an algorithm that determines for each gate the types of the offers used with this gate. Such an algorithm has not yet been developed for Atlantif.

- Another consequence of the typed ports in Fiacre is that for each gate G, all communication actions using G must have the same offer cardinality. It should be possible to overcome this restriction by the introduction of auxiliary constructs e.g., by extending all communication offers to a maximum.

- Offers in one communication action must not be mixed (i.e., containing emissions as well as receptions), because mixed offers are not supported by Fiacre.

- For each synchronizer G such that there exists a unit u which is in some, but not in all, synchronization sets of G (cf. the definition of $sync(G)$ on page 78), this unit must not be stopped by G. In our translator, the approaches applied to emulate different synchronization sets and to emulate unit stopping (cf. Section A.3.2) are similar; and because of this similarity they cannot be mixed. It is possible that other emulation approaches can avoid this problem.

A.3.4 Definition of the translator

We suppose an Atlantif module M defined by the units $u_1, \ldots, u_n$ and the synchronizers $G_1, \ldots, G_m$. The translation of M into a Fiacre program consists of the three following steps.

Step one: main component. The main component renames the auxiliary ports that will be defined in the component *l2* back to the identifers of the corresponding synchronizer identifiers, and it is itself defined as follows:

- The main component's identifier is the module M's identifier.
- For each visible (i.e., neither **hidden** nor **silent**) synchronizer G among $G_1, \ldots, G_m$, the list of ports in the header contains one port also named G declared with the according channel (i.e., the channel we identified for G during step one).
- The list of parameters is empty.
- For each **hidden** or **silent** synchronizer G among $G_1, \ldots, G_m$, the list of local port declarations contains one port also named G declared with the according channel. Thus, in an instanciation of the main component these ports will be hidden.
- The parallel composition operator is defined as follows: Let c_i (for each $i \in 1..m$) denote the cardinality of $\mathit{sync}(G_i)$. Then the parallel composition has the form
$$\textbf{par} \ast \textbf{ in } \mathit{l2}\ \texttt{[}\underbrace{G_1\texttt{,}\ldots\texttt{,}G_1}_{c_1 \text{ times}}\texttt{,}\ldots\texttt{,}\underbrace{G_m\texttt{,}\ldots\texttt{,}G_m}_{c_m \text{ times}}\texttt{]}\ \textbf{end par}\ .$$
Thus, ports in the header of *l2* that represent different synchronization sets of the same synchronizer are given the same name (the synchronizer's identifier) on this level.
- Other than local port declarations and the parallel composition, the component body is empty i.e., it does not contain any variable declarations or priorities, nor an initial statement.

Step two: synchronizer analysis and parallel composition. For each $i \in 1..m$, we determine by the multibranch transitions of $u_1, \ldots, u_n$ which types and numbers of offers are exchanged via G_i. According to the above restrictions, there exists a $t_i \in \mathbb{N}$ such that G_i always occurs with t_i integer type offers. Thus, the channel that applies for our translation of G_i will be, in the Fiacre notation, $\underbrace{\textbf{int\#}\ldots\textbf{\#int}}_{t_i \text{ times}}$ (or empty, if $t_i = 0$).

Moreover, we need to analyze for G_i and each $j \in 1..n$ how exactly the unit u_j interacts by G_i (i.e., if u_j is started and/or stopped by G_i and if u_j participates always, sometimes, or never in a synchronization on G_i). Nine cases are distinguished in this analysis, depending on which we define two functions *sync_card* and *usage*, both with domain $\mathcal{G} \times \mathbb{U}$ (the gates and the units in M) and having values respectively in $\mathbb{N}$ and a set of tags $\{\mathit{NOTHING}, \mathit{ONLY\_SYNC}, \mathit{SYNC\_N\_STOP}, \mathit{ASY\_TERM}, \mathit{SYNC\_N\_RESET}, \mathit{ASY\_RESET}, \mathit{STARTING}\}$.

(0) $u_j \notin \mathit{start}(G_i), u_j \notin \mathit{stop}(G_i), (\forall \mathcal{U} \in \mathit{sync}(G_i))\ u_j \notin \mathcal{U}$.
We define $\mathit{usage}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{NOTHING}$ and $\mathit{sync\_card}(G_i, u_j) \stackrel{\text{def}}{=} 0$.

(1) $u_j \in \mathit{start}(G_i), u_j \notin \mathit{stop}(G_i)$ (by static semantics: $(\forall \mathcal{U} \in \mathit{sync}(G_i))\ u_j \notin \mathcal{U}$).
We define $\mathit{usage}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{STARTING}$ and $\mathit{sync\_card}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{card}(\mathit{sync}(G_i))$.

(2) $u_j \notin \mathit{start}(G_i), u_j \notin \mathit{stop}(G_i), (\exists \mathcal{U} \in \mathit{sync}(G_i))\ u_j \in \mathcal{U}$.
We define $\mathit{usage}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{ONLY\_SYNC}$ and $\mathit{sync\_card}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{card}(\{\mathcal{U} \in \mathit{sync}(G_i) \mid u_j \in \mathcal{U}\})$.

(3) $u_j \notin \mathit{start}(G_i), u_j \in \mathit{stop}(G_i), (\forall \mathcal{U} \in \mathit{sync}(G_i))\ u_j \in \mathcal{U}$.
We define $\mathit{usage}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{SYNC\_N\_STOP}$
and $\mathit{sync\_card}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{card}(\mathit{sync}(G_i))$.

(4) $u_j \notin \mathit{start}(G_i), u_j \in \mathit{stop}(G_i), (\exists \mathcal{U}, \mathcal{U}' \in \mathit{sync}(G_i))\ u_j \in \mathcal{U} \wedge u_j \notin \mathcal{U}'$.
According to the restrictions listed above, this case is excluded.

(5) $u_j \notin \mathit{start}(G_i), u_j \in \mathit{stop}(G_i), (\forall \mathcal{U} \in \mathit{sync}(G_i))\ u_j \notin \mathcal{U}$.
We define $\mathit{usage}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{ASY\_TERM}$ and $\mathit{sync\_card}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{card}(\mathit{sync}(G_i))$.

(6) $u_j \in \mathit{start}(G_i), u_j \in \mathit{stop}(G_i), (\forall \mathcal{U} \in \mathit{sync}(G_i))\ u_j \in \mathcal{U}$.
We define $\mathit{usage}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{SYNC\_N\_RESET}$
and $\mathit{sync\_card}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{card}(\mathit{sync}(G_i))$.

(7) $u_j \in \mathit{start}(G_i), u_j \in \mathit{stop}(G_i), (\exists \mathcal{U}, \mathcal{U}' \in \mathit{sync}(G_i))\ u_j \in \mathcal{U} \wedge u_j \notin \mathcal{U}'$.
According to the restrictions listed above, this case is excluded.

(8) $u_j \in \mathit{start}(G_i), u_j \in \mathit{stop}(G_i), (\forall \mathcal{U} \in \mathit{sync}(G_i))\ u_j \notin \mathcal{U}$.
We define $\mathit{usage}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{ASY\_RESET}$ and $\mathit{sync\_card}(G_i, u_j) \stackrel{\text{def}}{=} \mathit{card}(\mathit{sync}(G_i))$.

The number $\mathit{sync\_card}(G_i, u_j)$ expresses how many synchronization sets of G_i concern the unit u_j. This does not always mean those synchronization sets contain u_j e.g., in the cases (1), (5), and (8), $\mathit{sync\_card}(G_i, u_j)$ is not zero although u_j is not in any synchronization set. Instead, we extend our perspective to the starting and/or stopping of u_j by G_i.

It is for this reason that we excluded the cases (4) and (7), because there u_j is concerned in two different ways by the synchronization sets: By some sets, it is synchronizing and stopped, by other sets it is only stopped.

Given this analysis of the synchronizers, we now define a structure that represents the composition. This structure will be given by a Fiacre *component* named "*l2*" (indicating that this component is situated on a second level above the processes) and defined as follows:

- The list of ports in the header contains for each $i \in 1..m$ a sucession of k (with $k = \mathit{card}(\mathit{sync}(G_i))$) ports $G_i\_l2\_1, \ldots, G_i\_l2\_k$ with the according channel. Given this, we have a one-to-one correspondance between all synchronization sets of M and the ports of *l2*, which was already prepared by the port parameters of the instanciation of *l2* in the main component.

- The list of parameters is empty.
- The component body does not contain any variable declarations, port declarations, or priorities, nor an initial statement.
- The parallel composition operator is defined as follows: Let for each $j \in 1..n$, $\text{port}^j_1, \ldots, \text{port}^j_{k_j}$ be those port identifiers (as defined in *l2*'s header) corresponding to synchronization sets concerning the unit u_j. Then the parallel composition has the form

  par
  port^1_1, ... ,$\text{port}^1_{k_1}$ `->` P_u [port^1_1, ... ,$\text{port}^1_{k_1}$]
  `|| ... ||`
  port^n_1, ... ,$\text{port}^n_{k_n}$ `->` P_u [port^n_1, ... ,$\text{port}^n_{k_n}$]
  end par.

Our construction of the parallel composition operator ensures that each process synchronizes on exactly those synchronization sets that concern the corresponding unit, either by being in the synchronization set or by being started and/or stopped by the corresponding synchronizer. Examples A.2 and A.3 will further illustrate this approach.

Step three: unit translations. For each $j \in 1..n$, the unit u_j is translated into one FIACRE process P_{u_j}. As described above in Section A.3.1, a process consists of six parts, which we define as follows:

- The list of ports used by the process contains one element for each synchronization set concerning u_j: For each $i \in 1..m$ such that $\mathit{sync\_card}(G_i, u_j) = m > 0$, we thus have the ports $G_i\_u_j\_1, \ldots, G_i\_u_j\_m$, each defined with the channel that applies for G_i.
- The list of data parameters is empty.
- The list of discrete state identifiers contains all discrete state identifiers occurring in u_j. Additionally, it contains a state "*stop_state*", if one or more of the following is true:
  - $(\exists i \in 1..n)\ \mathit{usage}(G_i, u_j) = \mathit{ASY\_TERM} \lor \mathit{usage}(G_i, u_j) = \mathit{SYNC\_N\_STOP}$ i.e., at least one synchronizer stops u_j without immediately restarting it.
  - u_j is not among the initial units of M.
  - One multibranch transition of u_j contains a **stop** action.
- The list of local variables contains all variables of u_j, declared with their types and, if applicable, their initial values.

- The initial statement of P_{u_j} is defined by a jump to the discrete state in which the process starts: If u_j is among the initial units of M, then the initial statement is "**init to** s_0", otherwise it is "**init to** *stop_state*" (where s_0 is the first discrete state of u_j).

- The list of multibranch transitions contains one element for each discrete state i.e., for multibranch transition "**from** s A" in u_j. Two cases have to be distinguished:

  Case one: There is no G such that $usage(G, u_j) \in \{ASY\_TERM, ASY\_RESET\}$. Then the corresponding multibranch transition in the process P_{u_j} is defined by

  "**from** s $ATLANTIF\_action\_to\_FIACRE(A)$"
  where $ATLANTIF\_action\_to\_FIACRE$ is a function defined in Fig. A.13.

  Case two: There are gates $G'_1, \ldots, G'_{k'}$ tagged with ASY_TERM and $G'_{k'+1}, \ldots, G'_k$ tagged with ASY_RESET ($k > 0$) for this unit. Then the corresponding multibranch transition in the process P_{u_j} is defined by

$$
\begin{array}{l}
\textbf{from } s \\
\ \textbf{select} \\
\quad G'_1\_u_j\_1 \ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_1 \text{ times}} ;\textbf{to } stop\_state \ \texttt{[]} \ldots \texttt{[]} \ G'_1\_u_j\_m_1 \ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_1 \text{ times}} ;\textbf{to } stop\_state \\
\quad\ \texttt{[]} \ \ldots \ \texttt{[]} \\
\quad G'_{k'}\_u_j\_1 \ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_{k'} \text{ times}} ;\textbf{to } stop\_state \ \texttt{[]} \ldots \texttt{[]} \ G'_{k'}\_u_j\_m_{k'} \ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_{k'} \text{ times}} ;\textbf{to } stop\_state \\
\quad\ \texttt{[]} \\
\quad G'_{k'+1}\_u_j\_1 \ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_{k'+1} \text{ times}} ;\textbf{to } s_0 \ \texttt{[]} \ldots \texttt{[]} \ G'_{k'+1}\_u_j\_m_{k'+1} \ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_{k'+1} \text{ times}} ;\textbf{to } s_0 \\
\quad\ \texttt{[]} \ \ldots \ \texttt{[]} \\
\quad G'_k\_u_j\_1 \ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_k \text{ times}} ;\textbf{to } s_0 \ \texttt{[]} \ldots \texttt{[]} \ G'_k\_u_j\_m_k \ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_k \text{ times}} ;\textbf{to } s_0 \\
\quad\ \texttt{[]} \ ATLANTIF\_action\_to\_FIACRE(A) \\
\ \textbf{end select}
\end{array}
$$

  where s_0 is the first discrete state of u_j and for each $i \in 1..k$, $m_i = sync\_card(G'_i, u_j)$ and o_i is the offer cardinality of G'_i.

  After translating the discrete states of u_j, the process P_{u_j} possibly also needs to be defined with the discrete state *stop_state* (if it is declared above). Then again, we have to distinguish two cases:

  Case one: u_j is not affected by synchronizations in other units i.e., there is no G such that $usage(G, u_j) \in \{ASY\_TERM, ASY\_RESET, STARTING\}$. Then the last multibranch transition in the process P_{u_j} is defined by "**from** *stop_state* **null**". This means that once P_{u_j} is in this state, it stays there.

  Case two: There are gates $G'_1, \ldots, G'_{k'}$ tagged with ASY_TERM and $G'_{k'+1}, \ldots, G'_k$ tagged with ASY_RESET or with $STARTING$ ($k > 0$) for this unit. Then the last multibranch transition in the process P_{u_j} is defined by

from *stop_state*
select
$G'_1\_u_j\_1\ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_1 \text{ times}}\ ;\textbf{to}\ \mathit{stop\_state}\ \ \texttt{[]} \ldots \texttt{[]}\ \ G'_1\_u_j\_m_1\ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_1 \text{ times}}\ ;\textbf{to}\ \mathit{stop\_state}$
[] ... []
$G'_{k'}\_u_j\_1\ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_{k'} \text{ times}}\ ;\textbf{to}\ \mathit{stop\_state}\ \ \texttt{[]} \ldots \texttt{[]}\ \ G'_{k'}\_u_j\_m_{k'}\ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_{k'} \text{ times}}\ ;\textbf{to}\ \mathit{stop\_state}$
[]
$G'_{k'+1}\_u_j\_1\ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_{k'+1} \text{ times}}\ ;\textbf{to}\ s_0\ \ \texttt{[]} \ldots \texttt{[]}\ \ G'_{k'+1}\_u_j\_m_{k'+1}\ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_{k'+1} \text{ times}}\ ;\textbf{to}\ s_0$
[] ... []
$G'_k\_u_j\_1\ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_k \text{ times}}\ ;\textbf{to}\ s_0\ \ \texttt{[]} \ldots \texttt{[]}\ \ G'_k\_u_j\_m_k\ ?\underbrace{\textbf{any}, \ldots, \textbf{any}}_{o_k \text{ times}}\ ;\textbf{to}\ s_0$
end select

where s_0 is the first discrete state of u_j and for each $i \in 1..k$, $m_i = \mathit{sync\_card}(G'_i, u_j)$ and o_i is the offer cardinality of G'_i.

The function *ATLANTIF_action_to_FIACRE*, defined in Fig. A.13 (supposing the context of a unit u with initial state s_0), translates ATLANTIF actions to FIACRE statements. It can be seen that most constructs are the same in both languages and remain unchanged. Four exceptions can be seen:

- The real-time syntax of ATLANTIF actions (which we excluded for this translation) is not translated.
- The **reset** action does not exist in FIACRE, thus is is translated by the **null** action.
- The **stop** action is translated like the syntax from which it is derived according to Remark 4.11. We used a similar approach in the translator to UPPAAL (cf. Section 6.1.3).
- Most importantly, the communication action is split up into several communication actions grouped by a **select** action, defining one communication for each synchronization set of G in which u occurs.

Examples. We now illustrate the translator by two examples, the first one of which demonstrates how the **among** synchronization is handled, the second of which concerns asynchronous termination.

Example A.2. *We place the simple example of a synchronizer with the* **among** *synchronization formula stated in Sections 4.5.1 and 5.5.2 in the context of a very simple* ATLANTIF *module, given in Fig. A.14.*

Following the rules given in this section, this module is translated[35] *to the* FIACRE *program shown in Fig. A.15.*

[35] We obtained the code of this translation, as well as in the following example, by our tool implementation, described in Section 6.4.

$$
\begin{array}{l}
\mathit{ATLANTIF\_action\_to\_FIACRE}(\mathbf{null}) \stackrel{\text{def}}{=} \mathbf{null} \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(V_0\,,\ldots,V_n \;\texttt{:=}\; E_0\,,\ldots,E_n) \stackrel{\text{def}}{=} V_0\,,\ldots,V_n \;\texttt{:=}\; E_0\,,\ldots,E_n \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(V_0\,,\ldots,V_n \;\texttt{:=}\; T_0\,,\ldots,T_n \;\mathbf{where}\; E) \stackrel{\text{def}}{=} V_0\,,\ldots,V_n \;\texttt{:=}\; \mathbf{where}\; E \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(\mathbf{reset}\; V_0\,,\ldots,V_n) \stackrel{\text{def}}{=} \mathbf{null} \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(G\ \texttt{?}P_1\ldots\texttt{?}P_n) \stackrel{\text{def}}{=} \\
\quad \begin{cases}
\mathbf{select}\; G\_u\_1\ \texttt{?}P_1\ldots P_n\,\texttt{;}\ \mathbf{to}\; \mathit{stop\_state}\ \texttt{[]}\ldots\texttt{[]} & \\
\qquad G\_u\_m\ \texttt{?}P_1\ldots P_n\,\texttt{;}\ \mathbf{to}\; \mathit{stop\_state}\ \mathbf{end\ select} & \text{if } \mathit{usage}(G,u) = \mathit{SYNC\_N\_STOP} \\
\mathbf{select}\; G\_u\_1\ \texttt{?}P_1\ldots P_n\,\texttt{;}\ \mathbf{to}\; s_0\ \texttt{[]}\ldots\texttt{[]} & \\
\qquad G\_u\_m\ \texttt{?}P_1\ldots P_n\,\texttt{;}\ \mathbf{to}\; s_0\ \mathbf{end\ select} & \text{if } \mathit{usage}(G,u) = \mathit{SYNC\_N\_RESET} \\
\mathbf{select}\; G\_u\_1\ \texttt{?}P_1\ldots P_n\ \texttt{[]}\ldots\texttt{[]} & \\
\qquad G\_u\_m\ \texttt{?}O_1\ldots O_n\ \mathbf{end\ select} & \text{otherwise}
\end{cases} \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(G\ \texttt{!}E_1\ldots\texttt{!}E_n) \stackrel{\text{def}}{=} \\
\quad \begin{cases}
\mathbf{select}\; G\_u\_1\ \texttt{!}E_1\ldots E_n\,\texttt{;}\ \mathbf{to}\; \mathit{stop\_state}\ \texttt{[]}\ldots\texttt{[]} & \\
\qquad G\_u\_m\ \texttt{!}E_1\ldots E_n\,\texttt{;}\ \mathbf{to}\; \mathit{stop\_state}\ \mathbf{end\ select} & \text{if } \mathit{usage}(G,u) = \mathit{SYNC\_N\_STOP} \\
\mathbf{select}\; G\_u\_1\ \texttt{!}E_1\ldots E_n\,\texttt{;}\ \mathbf{to}\; s_0\ \texttt{[]}\ldots\texttt{[]} & \\
\qquad G\_u\_m\ \texttt{!}E_1\ldots E_n\,\texttt{;}\ \mathbf{to}\; s_0\ \mathbf{end\ select} & \text{if } \mathit{usage}(G,u) = \mathit{SYNC\_N\_RESET} \\
\mathbf{select}\; G\_u\_1\ \texttt{!}E_1\ldots E_n\ \texttt{[]}\ldots\texttt{[]} & \\
\qquad G\_u\_m\ \texttt{!}O_1\ldots O_n\ \mathbf{end\ select} & \text{otherwise}
\end{cases} \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(\mathbf{to}\; s') \stackrel{\text{def}}{=} \mathbf{to}\; s' \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(\mathbf{stop}) \stackrel{\text{def}}{=} \mathbf{to}\; \mathit{stop\_state} \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(A_1\,\texttt{;}\,A_2) \stackrel{\text{def}}{=} \\
\quad \mathit{ATLANTIF\_action\_to\_FIACRE}(A_1)\,\texttt{;}\ \mathit{ATLANTIF\_action\_to\_FIACRE}(A_2) \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(\mathbf{select}\; A_0\ \texttt{[]}\ldots\texttt{[]}\ A_n\ \mathbf{end}) \stackrel{\text{def}}{=} \\
\quad \mathbf{select}\; \mathit{ATLANTIF\_action\_to\_FIACRE}(A_0)\ \texttt{[]}\ldots\texttt{[]} \\
\qquad \mathit{ATLANTIF\_action\_to\_FIACRE}(A_n)\ \mathbf{end\ select} \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(\mathbf{case}\; E_0\ \mathbf{is}\ P_0\texttt{->}A_0\,\texttt{|}\ldots\texttt{|}\,P_n\texttt{->}A_n\ \mathbf{end}) \stackrel{\text{def}}{=} \\
\quad \mathbf{case}\; E_0\ \mathbf{of}\ P_0\texttt{->}\mathit{ATLANTIF\_action\_to\_FIACRE}(A_0)\,\texttt{|}\ldots\texttt{|} \\
\qquad P_n\texttt{->}\mathit{ATLANTIF\_action\_to\_FIACRE}(A_n)\ \mathbf{end\ case} \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(\mathbf{if}\; E_0\ \mathbf{then}\ A_1\ \mathbf{else}\ A_2\ \mathbf{end}) \stackrel{\text{def}}{=} \\
\quad \mathbf{if}\; E_0\ \mathbf{then}\ \mathit{ATLANTIF\_action\_to\_FIACRE}(A_1) \\
\qquad \mathbf{else}\ \mathit{ATLANTIF\_action\_to\_FIACRE}(A_2)\ \mathbf{end\ if} \\
\mathit{ATLANTIF\_action\_to\_FIACRE}(\mathbf{while}\; E_0\ \mathbf{do}\ A\ \mathbf{end}) \stackrel{\text{def}}{=} \\
\quad \mathbf{while}\; E_0\ \mathbf{do}\ \mathit{ATLANTIF\_action\_to\_FIACRE}(A)\ \mathbf{end\ while}
\end{array}
$$

Figure A.13: Function *ATLANTIF_action_to_FIACRE*

```
module Generalized_Parallel_Composition is
  no time

sync G is 2 or 3 among (B1, B2, B3) end sync
init B1, B2, B3

unit B1 is
  from S1
    G; stop
end unit

unit B2 is
  from S2
    G; stop
end unit

unit B3 is
  from S3
    G; stop
end unit
end module
```

Figure A.14: 2 **among** 3 synchronization formula in the context of an ATLANTIF module

The synchronizer G of the ATLANTIF *module has four synchronization sets, each of which gets one port associated in the component l2: The set* $\{B1, B2\}$ *corresponds to the port G_l2_1, the set* $\{B1, B3\}$ *corresponds to the port G_l2_2, the set* $\{B2, B3\}$ *corresponds to the port G_l2_3, and the set* $\{B1, B2, B3\}$ *corresponds to the port G_l2_4. Therefore, the port G_l2_1 appears in the synchronizing ports and in the parameters associated to the processes B1 and B2, the port G_l2_2 appears in the synchronizing ports and in the parameters associated to the processes B1 and B3, etc.*

As each unit belongs to three different synchronization sets of G, each process is parameterized by three ports derived from G. Then, each commuication action by G in a multibranch transition is translated by a choice over communications by these three ports.

Example A.3. *This example demonstrates the translation of aynchronous termination. The* ATLANTIF *module we translate is a slight variation of that shown in Fig. 5.9 on page 119: According to the restrictions on the translator, we suppose this module to be untimed, thus we remove the two* **wait** *actions.*

Then, we obtain the translation shown in Fig. A.16.

Although in the ATLANTIF *module, the units U1 and U2 are not in the synchronization set of RedButton, their process translations do synchronize on the attendant port to emulate their asynchronus termination. Therefore, each discrete state in these processes contains the possibility to synchronize on RedButton i.e., such a synchronization is always possible.*

A.3.5 Discussion

Impact of the restrictions

Clearly, the most important restriction is the limitation to untimed ATLANTIF, because real time played a central role in the definition of ATLANTIF and most examples in this thesis contain real-time syntax. Nevertheless, sometimes real time can be abstracted from, thus for certain verification problems an untimed translator can still be useful.

```
process B1 [G_B1_1, G_B1_2,
        G_B1_3 : none] is
states S1, stop_state
init to S1
  from S1
    select
      G_B1_1
    []
      G_B1_2
    []
      G_B1_3
    end select;
    to stop_state
  from stop_state
    null

process B2 [G_B2_1, G_B2_2,
        G_B2_3 : none] is
states S2, stop_state
init to S2
  from S2
    select
      G_B2_1
    []
      G_B2_2
    []
      G_B2_3
    end select;
    to stop_state
  from stop_state
    null
```

```
process B3 [G_B3_1, G_B3_2,
        G_B3_3 : none] is
states S3, stop_state
init to S3
  from S3
    select
      G_B3_1
    []
      G_B3_2
    []
      G_B3_3
    end select;
    to stop_state
  from stop_state
    null

component l2 [G_l2_1, G_l2_2,
        G_l2_3, G_l2_4 : none] is
  par G_l2_1, G_l2_2, G_l2_4 ->
      B1 [G_l2_1, G_l2_2, G_l2_4]
    || G_l2_1, G_l2_3, G_l2_4 ->
      B2 [G_l2_1, G_l2_3, G_l2_4]
    || G_l2_2, G_l2_3, G_l2_4 ->
      B3 [G_l2_2, G_l2_3, G_l2_4]
  end par

component Generalized_Parallel_Composition
             [G : none] is
  par * in l2 [G, G, G, G] end par

Generalized_Parallel_Composition
```

Figure A.15: Generated Fiacre program for the module of Fig. A.14

```
process U1 [G_U1_1 : none,
        RedButton_U1_1 : none] is
states S1, stop_state
init to S1
  from S1
    select
      RedButton_U1_1; to stop_state
    []
      G_U1_1; to S1
    end select
  from stop_state
    select
      RedButton_U1_1; to stop_state
    end select

process U2 [G_U2_1 : none,
        RedButton_U2_1 : none] is
states S2, stop_state
init to S2
  from S2
    select
      RedButton_U2_1; to stop_state
    []
      G_U2_1; to S2
    end select
  from stop_state
    select
      RedButton_U2_1; to stop_state
    end select

process Supervisor
     [RedButton_Supervisor_1 : none] is
states S3, stop_state
init to S3
  from S3
    RedButton_Supervisor_1;
    to stop_state
  from stop_state
    null

component l2 [G_l2_1 : none,
        RedButton_l2_1 : none] is
  par G_l2_1, RedButton_l2_1 ->
      U1 [G_l2_1, RedButton_l2_1]
    || G_l2_1, RedButton_l2_1 ->
      U2 [G_l2_1, RedButton_l2_1]
    || RedButton_l2_1 ->
      Supervisor [RedButton_l2_1]
  end par

component Asynchronous_Termination
     [G : none, RedButton : none] is
  par * in l2 [G, RedButton] end par

Asynchronous_Termination
```

Figure A.16: Generated FIACRE program for untimed asynchronous termination

Moreover, one of our initial intentions in the definition of a translator to Fiacre was to create a link to the Cadp toolbox via the "flac" tool and the Lotos language. These tools do not provide real-time aspects, thus a translation of them would not be of use to us.

Imprecision problems of the translator

Again (as in the translators to Uppaal and to Tina), execution paths (of a multibranch transition) without communication action and synchronizations on silent synchronizers produce unlabelled transitions; also **reset** actions are not represented correctly.

Apart from this, however, no other imprecisions seem to be produced by our translator, although no formal proof of correctness is possible in the moment, as the complete formal semantics of Fiacre has not yet been published.

Appendix B

Additional examples

B.1 Application of the generalized parallel composition

In Section 5.5, we provided general rules on how synchronization vectors and the generalized parallel composition operator can be represented in ATLANTIF. In this section, we will provide a further illustration of such a representation, by the example of the *Open Distributed Processes* (ODP) protocol, which we borrowed from [64] (i.e., the paper that defines the generalized parallel composition operator).

We suppose an environment composed of several objects, each of which can provide different services to other objects and/or make use a service provide by another object. An object providing a service is called a *server*; an object making use of a service is called a *client*. The coordination between the objects is assured by a "*trader*", which manages a database containing the information about which object offers which service.

The ATLANTIF code given in Fig. B.1 shows the most important ideas of a representation of ODP.

The four type definitions are not detailed: The type *Object* describes a subset of the natural integers, containing the numbers between 1 and n, where n is the number of objects. The type *Service* describes an enumeration of all services that can be provided by the servers. The type *DataBase* describes a set of tuples of one object and one service each, which expresses that the service is provided by the object. The type *Communication_Tag* describes an enumeration of two values only: "request" / "reply".

Moreover, two functions that are not detailed: The function $\mathit{Add\_To\_DataBase}(d, j, s)$ returns a *DataBase* consisting of a *DataBase* d extended with the tuple (j, s). The function $\mathit{Find\_In\_DataBase}(d, s)$ returns an object identifier j that is listed in the *DataBase* d to provide the service s. For simplicity, we assume that for all requests carried out by the function there is indeed an appropriate server.

Three gates are used: A synchronization on the gate E ("*export*") corresponds to a server object that wishes his service offer to be included in the trader's database. A synchro-

```
module Open_Distributed_Processes is
no time

type Object (...)
type Service (...)
type DataBase (...)
type Communication_Tag (...)

function Add_To_DataBase (...)
function Find_In_DataBase (...)

sync E is Trader and
    1 among(Object1,..., Objectn) end sync
sync I is Trader and
    1 among(Object1,..., Objectn) end sync
sync W is 2 among(Object1,..., Objectn) end sync

unit Trader is
variables d:DataBase:=empty,
          j:Object,
          s:Service
  from Ready
    select
        E ?j ?s;
        d:=Add_To_DataBase(d,j,s);
        to Ready
      []
        I ?j ?"request" ?s;
        to Search_Server
    end select
  from Search_Server
    I !j !"reply" !(Find_In_DataBase(d,s));
    to Ready
end unit

unit Object1 is
variables j:Object,
          s_1:Service
  from Init
    select to Export_Service
        [] to Import_Service end select
  from Export_Service
    E !1 !s_1;
    to Work_As_Server
  from Work_As_Server
    W ?1 ?j !s_1 ...;
    to Init
  from Import_Service
    I !1 !"request" !s_1;
    to Receive_Server
  from Receive_Server
    I ?1 ?"reply" ?j;
    to Work_As_Client
  from Work_AsClient
    W !j !1 !s_1 ...;
    to Init
end unit

(...)

unit Objectn is
variables j:Object,
          s_n:Service
  from Init
    select to Export_Service
        [] to Import_Service end select
  from Export_Service
    E !n !s_n;
    to Work_As_Server
  from Work_As_Server
    W ?n ?j !s_n ...;
    to Init
  from Import_Service
    I !n !"request" !s_n;
    to Receive_Server
  from Receive_Server
    I ?n ?"reply" ?j;
    to Work_As_Client
  from Work_AsClient
    W !j !n !s_n ...;
    to Init
end unit
end module
```

Figure B.1: ATLANTIF code for Open Distributed Processes nization on the gate I ("*import*") corresponds to a client object seeking information from the trader's database about a server for a given service. A synchronization on the gate W ("*work*") corresponds to a server object and a client object working together by a given service.

In the synchronizer definitions, it can be noticed that the **among** construct of the synchornizer formulas enables a very short and intuitive description of the possible synchronizations.

The code of the units representing the *Trader* and the n *Object*s is largely self-explanatory; while it should be noted that, for generality, each *Object* can act as a client as we as a server. Like in [64], we consider that for each requested service, a server is available. Note also that the *import* of a service is represented by two successive synchronizations (the first of which corresponding to the client requesting the trader for a service, the second corresponding to the trader's answer), which is also the structure the ODP example has in [64]. In particular, both communications use different offer profiles.

B.2 Timed semantics in synchronization chains

This example is designed to illustrate how phases are used during the elemination of **silent** synchronizations, more specifically **silent** synchronizations that start and stop units. Our intention thus being linked to the ATLANTIF semantics, the syntax definition of the module will sometimes be a bit artificial.

We consider a system describing an assembly line for toy cars, shown in Fig. B.2. One subsystem represents the beginning of the line (unit BB, "begin belt"), where two seconds elapse before the car parts begin to arrive on the belt. Until all parts arrive (gate PC, "place car parts"), up to five more seconds elapse. Then, the control is passed to the two subunits, which represent robot arms (gate BM, "begin manipulation").

Units $R1$ and $R2$ ("robot arm 1" and "robot arm 2") need five and three seconds respectively before they finish their manipulations (gate *FM*), which triggers a control passage to the system representing the end of the belt.

This subsystem (unit *EB*) has to wait 4 seconds until the assembled car arrives in its reach, then it needs exactly one second to handle it over to the environment (gate EC, "exit car"), which also triggers a control passage to the beginning of the line again.

Note that we shortened the names of units and synchronizers to ensure compact derivations.

Our illustration of the semantics by this example consists of the derivation of four TLTS transitions of this module. We show that from the initial state (which is $(\mathcal{U}_0, \pi_1, \theta_0, \rho_1) = (\{BB\}, [BB \mapsto Ry], [BB \mapsto (0, \mathbf{f})], [V \mapsto 2])$, where, as in Section 4.6.3, $\mathbf{f}$ and $\mathbf{t}$ are shorthands for **false** and bf true) 4 seconds may elapse before a τ-transition occurs, then 11 more seconds may elapse before a transition labeled EC occurs.

Formally:

```
module Toy_Car_Assembly_Line is
dense time
sync PC : hidden is BB end sync
sync BM : silent is BB
      stop BB
      start R1,R2
end sync
sync FM : silent is R1 and R2
      stop R1,R2,EB
      start EB
end sync
sync EC is EB stop EB
      start BB end sync

init BB (* initially started unit *)

unit BB is
   variables V : int := 2
   from Ry
      wait 2;PC in [0,5];to Ar
   from Ar
      wait 1;BM ;stop

   unit R1 is (* subunit of BB *)
      from Ac
         wait 5;FM ;stop
   end unit (* R1 *)

   unit R2 is (* subunit of BB *)
      from Ac
         wait 3;to A2
      from A2
         FM ;stop
   end unit (* R2 *)

end unit (* BB *)

unit EB is
   from Fi
      wait 4;EC in [1,1];stop
   end unit

end module
```

Figure B.2: ATLANTIF program for semantics example

$$(\mathcal{U}_0, \pi_1, \theta_0, \rho_1) \xrightarrow{4} (\mathcal{U}_0, \pi_1, \theta_0 + 4, \rho_1) \xrightarrow{\tau} (\mathcal{U}_0, \pi_2, \theta_0, \rho_1) \xrightarrow{11} (\mathcal{U}_0, \pi_2, \theta_0 + 11, \rho_1) \xrightarrow{EC} (\mathcal{U}_0, \pi_1, \theta_0, \rho_1),$$

where $\pi_2 = [BB \mapsto Ar]$.

First, $(\mathcal{U}_0, \pi_1, \theta_0, \rho_1) \xrightarrow{4} (\mathcal{U}_0, \pi_1, \theta_0 + 4, \rho_1)$ comes from the following derivation:

$$\frac{4 > 0 \wedge (\forall t' < 4)\ \mathit{relaxed}((\mathcal{U}_0, \pi_1, \theta_0 + t', \rho_1))}{(\mathcal{U}_0, \pi_1, \theta_0, \rho_1) \xrightarrow{4} (\mathcal{U}_0, \pi_1, \theta_0 + 4, \rho_1)}(\mathit{time})$$

Second, $(\mathcal{U}_0, \pi_1, \theta_0 + 4, \rho_1) \xrightarrow{\tau} (\mathcal{U}_0, \pi_2, \theta_0, \rho_1)$ comes from:

$$\frac{\mathit{enabled}((\mathcal{U}_0, \pi_1, \theta_0 + 4, \rho_1), PC, \mathbf{f}, (\mathcal{U}_0, \pi_2, \theta_0, \rho_1))}{(\mathcal{U}_0, \pi_1, \theta_0 + 4, \rho_1) \xrightarrow{\tau} (\mathcal{U}_0, \pi_2, \theta_0, \rho_1)}(\mathit{rdv})$$

where $\mathit{enabled}((\mathcal{U}_0, \pi_1, \theta_0 + 4, \rho_1), PC, \mathbf{f}, (\mathcal{U}_0, \pi_2, \theta_0, \rho_1))$, because $\{BB\} \in \mathit{sync}(PC)$, and $\{BB\} \subseteq \mathcal{U}_0$, and $(\mathit{act}(Ry), (4, \mathbf{f}), \rho_1) \overset{PC}{\Longrightarrow} (Ar, (2, \mathbf{f}), \rho_1)$, and $Ar \neq \delta$, and the evaluation of the three auxiliary predicates results in π_2, θ_0, and ρ_1 respectively.

The premise $(\mathit{act}(Ry), (4, \mathbf{f}), \rho_1) \overset{PC}{\Longrightarrow} (Ar, (2, \mathbf{f}), \rho_1)$ comes from the following, recalling that $\mathit{act}(Ry) =$ "**wait** 2; PC **in** [0, 5]; **to** Ar":

$$\frac{\dfrac{\mathit{eval}(2, \rho_1, 2) \wedge 4 \geq 2}{(\mathbf{wait}\ 2, (4, \mathbf{f}), \rho_1) \overset{\varepsilon}{\Longrightarrow} (\delta, (2, \mathbf{f}), \rho_1)}(\mathit{wait}) \quad (PC\ \mathbf{in}\ \texttt{[0,5]}\ \texttt{;}\mathbf{to}\ Ar, (2, \mathbf{f}), \rho_1) \overset{PC}{\Longrightarrow} (Ar, (2, \mathbf{f}), \rho_1)}{(\mathit{act}(Ry), (4, \mathbf{f}), \rho_1) \overset{PC}{\Longrightarrow} (Ar, (2, \mathbf{f}), \rho_1)}(\mathit{seq}_1)$$

At last, the premiss $(PC\ \mathbf{in}\ \texttt{[0,5]}\ \texttt{;}\mathbf{to}\ Ar, (2, \mathbf{f}), \rho_1) \overset{PC}{\Longrightarrow} (Ar, (2, \mathbf{f}), \rho_1)$ comes from:

$$\frac{\dfrac{\mathit{win\_eval}(\texttt{[0,5]}, \rho_1, [0, 5]) \wedge 2 \in [0, 5]}{(PC\ \mathbf{in}\ \texttt{[0,5]}, (2, \mathbf{f}), \rho_1) \overset{PC}{\Longrightarrow} (\delta, (2, \mathbf{f}), \rho_1)}(\mathit{comm}) \quad \dfrac{}{(\mathbf{to}\ Ar, (2, \mathbf{f}), \rho_1) \overset{\varepsilon}{\Longrightarrow} (Ar, (2, \mathbf{f}), \rho_1)}(\mathit{to})}{(PC\ \mathbf{in}\ \texttt{[0,5]}\ \texttt{;}\mathbf{to}\ Ar, (2, \mathbf{f}), \rho_1) \overset{PC}{\Longrightarrow} (Ar, (2, \mathbf{f}), \rho_1)}(\mathit{seq}_1)$$

The third transition $(\mathcal{U}_0, \pi_2, \theta_0, \rho_1) \xrightarrow{11} (\mathcal{U}_0, \pi_2, \theta_0 + 11, \rho_1)$ is derived similarly as the first transition.

Fourth, $(\mathcal{U}_0, \pi_2, \theta_0 + 11, \rho_1) \xrightarrow{EC} (\mathcal{U}_0, \pi_1, \theta_0, \rho_1)$ is derived from the predicate

$$\mathit{enabled}((\mathcal{U}_0, \pi_2, \theta_0 + 11, \rho_1), EC, \mathbf{f}, (\mathcal{U}_0, \pi_1, \theta_0, \rho_1)).$$

This predicate itself describes a chain of synchronizations and is derived from the three following predicates:

$$\mathit{synchronizing}(((\mathcal{U}_0, \pi_2, \theta_0 + 11, \rho_1), \varnothing), BM, \mathbf{f}, ((\{R1, R2\}, [R1 \mapsto Ac, R2 \mapsto Ac], [R1 \mapsto (10, \mathbf{f}), R2 \mapsto (10, \mathbf{f})], \rho_1), \{\{R1, R2\}\}))$$

$$\mathit{synchronizing}(((\{R1, R2\}, [R1 \mapsto Ac, R2 \mapsto Ac], [R1 \mapsto (10, \mathbf{f}), R2 \mapsto (10, \mathbf{f})], \rho_1), \{\{R1, R2\}\}), FM, \mathbf{f}, ((\{EB\}, [EB \mapsto Fi], [EB \mapsto (5, \mathbf{f})], \varnothing)\{EB\}))$$

$$synchronizing(((\{EB\}, [EB \mapsto Fi], [EB \mapsto (5, \mathbf{f})], \varnothing), \{\{EB\}\}), EC, \mathbf{f}, ((\mathcal{U}_0, \pi_1, \theta_0, \rho_1), \varnothing))$$

We take a closer look at the first of this three. It is derived with $\{BB\} \in sync(BM)$, and $\{BB\} \subseteq \mathcal{U}_0$, and $\{R1, R2\} = (\{BB\} \setminus stop(BM)) \cup start(BM)$, and $\mu = \mathbf{f}$, and $(act(Ar), \theta_0 + 11, \rho_1) \overset{BM}{\Longrightarrow} (\Omega, \theta_0 + 10, \rho_1)$, and $\Omega \neq \delta$. Furthermore,

$next\_\pi(\pi_2, [BB \mapsto \Omega], BM, [R1 \mapsto Ac, R2 \mapsto Ac])$
$next\_\theta(\theta_0 + 11, \mathcal{U}_0, 10, BM, [R1 \mapsto (10, \mathbf{f}), R2 \mapsto (10, \mathbf{f})])$
$next\_\rho(\rho_1, \rho_1, \{R1, R2\}, BM, \rho_1)$
(because V is accessible both in $R1$ and $R2$)
$next\_\alpha(\varnothing, \{BB\}, BM, \{\{R1, R2\}\})$
(because $(\varnothing \setminus \{BB\}) \cup \{(\{BB\} \setminus \{BB\}) \cup \{R1, R2\}\} = \{\{R1, R2\}\})$

In the second of the three synchronizations, the predicate $next\_\theta$ occurs as follows:

$$next\_\theta([R1 \mapsto (10, \mathbf{f}), R2 \mapsto (10, \mathbf{f})], \{R1, R2\}, 5, FM, [EB \mapsto (5, \mathbf{f})])$$

By the two local transitions on which this synchronization is based, the phases 5 and 7 respectively are reached. As the gate *FM* is **silent**, the new phase is calculated to be the minimum among those two values, thus $t_0 = 5$. Therefore, the newly started unit *EB* has then the time distribution $(5, \mathbf{f})$. This further illustrates the intuition behind the **silent** case in the definition of $next\_\theta$: Unit $R1$ needs 5 seconds to become ready to synchronize, unit $R2$ only needs 3. Thus $R2$ has to wait two seconds for $R1$, thus the phase of $R1$ is decisive.

Note that in no state of the TLTS, other units than *BB* are active. This can already be seen from the synchronizers, because all non-**silent** synchronizations "end" in *BB*.

B.3 Lamp

In Section 6.4, we make use of a simplified version of the light switch example of Fig. 4.1 on page 55. This simplified version is given here, in Fig. B.3.

```
module Light is dense time
sync Push is User and Lamp end sync
init User, Lamp
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 B.3: ATLANTIF module describing a light switch

Appendix C

Complete syntax

In Chapter 4, we defined the syntax of ATLANTIF in a variant of EBNF designed to be easily human-readable (by using bold fonts, truetype fonts, etc.). In the implementation of the `atlantif` tool, we defined the syntax by the variant of EBNF used in the `syntax` tool, which we used.

Thus, to complement our definitions of Table 4.3, we show here the complete implemented syntax. This also provides information on the syntax of constants, external types, and external functions, which we used in Chapter 7.

ATLANTIF grammar

```
<specification> = <module>;

<module> = "module" <identifier> "is"
   <timing-option>
   <types-functions-constants>
   <sync-declarations>
   <initial-state>
   <unit-declarations>
   "end" <module-end-tag>;

<timing-option> = ;
<timing-option> = "no" "time";
<timing-option> = "dense" "time";
<timing-option> = "discrete" "time";

<initial-state> = "init" <identifier-list>;

<module-end-tag> = ;
<module-end-tag> = "module";

*=============================================================================

<types-functions-constants> = ;
<types-functions-constants> = <type-declaration> <types-functions-constants> ;
```

```
<types-functions-constants> = <function-declaration> <types-functions-constants> ;
<types-functions-constants> = <constant-declarations> <types-functions-constants> ;

<type-declaration> = "type" <identifier> "is" "!" "external" <qidentifier>
     "end" "type" ;

<type> = "time" ;
<type> = "int" ;
<type> = "bool" ;
<type> = <identifier> ;

<function-declaration> = "function" <identifier>
 "(" <parameters> ")"
 ":" <type>
 "is" <function-body>
 "end" "function" ;

<function-body> = <function-variables> <action> "return" <expression> ;
<function-body> = "!" "external" <qidentifier> ;

<parameters> = ;
<parameters> = <variable-list> ;

<function-variables> = ;
<function-variables> = "variables" <variable-list> ;

<constant-declarations> = "constant" <constant-declaration-list>;

<constant-declaration-list> = <constant-declaration>;
<constant-declaration-list> = <constant-declaration> "," <constant-declaration-list>;

<constant-declaration> = <identifier> ":" <type> "is" <expression>;

<sync-declarations> = ;
<sync-declarations> = <sync-list>;

<sync-list> = <sync>;
<sync-list> = <sync> <sync-list>;

<sync> = "sync" <identifier> <rdv-gate> "is"
 <unit-combination> <opt-disabling> <opt-activation>
 "end" "sync";

<unit-combination-list> = <unit-combination>;
<unit-combination-list> = <unit-combination> "," <unit-combination-list>;

<unit-combination> = <unit-combination-2>;
<unit-combination> = <unit-combination> "or" <unit-combination-2>;

<unit-combination-2> = <unit-combination-3>;
<unit-combination-2> = <unit-combination-2> "and" <unit-combination-3>;

<unit-combination-3> = <identifier>;
<unit-combination-3> = "(" <unit-combination> ")";
```

```
<unit-combination-3> = <natural-list> "among" "(" <unit-combination-list> ")";

<natural-list> = %NATNUMBER ;
<natural-list> = %NATNUMBER "or" <natural-list>;

<opt-disabling> = ;
<opt-disabling> = "stop" <identifier-list>;

<opt-activation> = ;
<opt-activation> = "start" <identifier-list>;

<rdv-gate> = ;
<rdv-gate> = ":" "silent";
<rdv-gate> = ":" "hidden";
<rdv-gate> = ":" "urgent";
<rdv-gate> = ":" "visible";

<unit-declarations> = <unit-declaration>;
<unit-declarations> = <unit-declaration> <unit-declarations>;

<unit-declaration> = "unit" <identifier> "is"
     <opt-var-declaration> <opt-transition-list>
     <opt-subunit-declarations>
     "end" "unit";

<opt-var-declaration> = ;
<opt-var-declaration> = "variables" <variable-list> ;

<opt-subunit-declarations> = ;
<opt-subunit-declarations> = <unit-declarations> ;

*============================================================================

<condition> = ;
<condition> = "where" <expression> ;

*============================================================================

<variable-list> = <variable-declaration> ;
<variable-list> = <variable-declaration> "," <variable-list> ;

<variable-declaration> = <identifier> ":" <type> ;
<variable-declaration> = <identifier> ":" <type> ":=" <expression>;

*============================================================================

<opt-transition-list> = ;
<opt-transition-list> = <transition-list>;

<transition-list> = <transition>;
<transition-list> = <transition> <transition-list> ;

<transition> = "from" <identifier> <action> ;
```

```
<action> = <single-action> ;
<action> = <single-action> ";" <action> ;

<single-action> = "null" ;
<single-action> = "stop" ;
<single-action> = "wait" <expression> ;
<single-action> = <identifier-list> ":=" <assignment> ;
<single-action> = "reset" <identifier-list> ;
<single-action> = <identifier> <offer-list> <timed-condition>;
<single-action> = "to" <identifier> ;
<single-action> = "select" <select-action> "end" <select-end-tag> ;
<single-action> = "case" <expression> "is" <case-rule-list>
  "end" <case-end-tag> ;
<single-action> = "while" <expression> "do" <action> "end" <while-end-tag> ;
<single-action> = "if" <expression> "then" <action> <else-branch> "end" <if-end-tag> ;
<single-action> = "for" <identifier> "in" <expression> %DOTDOT <expression>
  "do" <action> "end" <for-end-tag> ;

<else-branch> = ;
<else-branch> = "else" <action> ;

<select-end-tag> = ;
<select-end-tag> = "select" ;

<case-end-tag> = ;
<case-end-tag> = "case" ;

<while-end-tag> = ;
<while-end-tag> = "while" ;

<if-end-tag> = ;
<if-end-tag> = "if" ;

<for-end-tag> = ;
<for-end-tag> = "for" ;

<assignment> = <expression-list> ;
<assignment> = "any" <type-list> <condition> ;

<type-list> = <type> ;
<type-list> = <type> "," <type-list> ;

<timed-condition> = ;
<timed-condition> = "in" <intervals> ;
<timed-condition> = "may" "in" <intervals> ;
<timed-condition> = "must" "in" <intervals> ;

<intervals> = <intervals-2> ;
<intervals> = <intervals> "or" <intervals-2> ;

<intervals-2> = <intervals-3> ;
<intervals-2> = <intervals-2> "and" <intervals-3> ;

<intervals-3> = "[" <expression> "," <expression> "]" ;
```

```
<intervals-3> = "]" <expression> "," <expression> "]" ;
<intervals-3> = "[" <expression> "," <expression> "[" ;
<intervals-3> = "]" <expression> "," <expression> "[" ;
<intervals-3> = "[" <expression> "," "..." "[" ;
<intervals-3> = "]" <expression> "," "..." "[" ;
<intervals-3> = "(" <intervals> ")" ;

<offer-list> = ;
<offer-list> = <offer> <offer-list> ;

<offer> = "!" <expression> ;
<offer> = "?" <pattern> ;

<select-action> = <action> ;
<select-action> = <action> "[]" <select-action> ;

<case-rule-list> = <case-rule> ;
<case-rule-list> = <case-rule> "|" <case-rule-list> ;

<case-rule> = <pattern-list> <condition> "->" <action> ;

<pattern-list> = <pattern-tuple> ;
<pattern-list> = <pattern-tuple> "|" <pattern-list> ;

<pattern-tuple> = <pattern> ;
<pattern-tuple> = <pattern> "," <pattern-tuple> ;

<pattern> = <identifier> ;
<pattern> = %FLOATNUMBER ;
<pattern> = "- " %FLOATNUMBER ;
<pattern> = %NATNUMBER ;
<pattern> = "- " %NATNUMBER ;
<pattern> = "true" ;
<pattern> = "false" ;
<pattern> = <identifier> "(" <pattern-tuple> ")" ;
<pattern> = "(" <pattern> <condition> ")" ;
<pattern> = "any" <type> ;

*========================================================================

<expression-list> = <expression> ;
<expression-list> = <expression> "," <expression-list> ;

<expression> = <level-1-expression> ;
<expression> = <level-1-expression> <level-0-infix> <level-1-expression> ;

<level-1-expression> = <level-2-expression> ;
<level-1-expression> = <level-1-expression> <level-1-infix> <level-2-expression> ;

<level-2-expression> = <level-3-expression> ;
<level-2-expression> = <level-2-expression> <level-2-infix> <level-3-expression> ;

<level-3-expression> = <level-4-expression> ;
<level-3-expression> = <unary-prefix> <level-4-expression> ;
```

```
<level-4-expression> = <identifier> ;
<level-4-expression> = <identifier> <call-parameters> ;
<level-4-expression> = <constant> ;
<level-4-expression> = <level-4-expression> "." "[" <expression> "]" ;
<level-4-expression> = <level-4-expression> "." <identifier> ;
<level-4-expression> = "(" <expression> ")" ;

<level-0-infix> = "=" ;
<level-0-infix> = "==" ;
<level-0-infix> = "!=" ;
<level-0-infix> = "<" ;
<level-0-infix> = ">" ;
<level-0-infix> = "<=" ;
<level-0-infix> = ">=" ;

<level-1-infix> = "+" ;
<level-1-infix> = "- " ;
<level-1-infix> = "or" ;

<level-2-infix> = "*" ;
<level-2-infix> = "/" ;
<level-2-infix> = "%" ;
<level-2-infix> = "and" ;

<unary-prefix> = "+" ;
<unary-prefix> = "- " ;
<unary-prefix> = "not" ;

<constant> = %FLOATNUMBER ;
<constant> = %NATNUMBER ;
<constant> = "true" ;
<constant> = "false" ;
<constant> = "infinity" ;

<call-parameters> = "(" ")" ;
<call-parameters> = "(" <expression-list> ")" ;

*============================================================================

<identifier> = %IDENTIFIER ;

<qidentifier> = %QIDENTIFIER ;

<identifier-list> = <identifier> ;
<identifier-list> = <identifier> "," <identifier-list> ;
```

The terminal symbols used in the preceeding definition are either string constants or among the following:

- "`%IDENTIFIER`" is a string composed of latin letters, digits from 0 to 9, and underscores. It begins with a letter and does not end with an underscore.

- "`%QIDENTIFIER`" is a string beginning and ending with double quotes, and containing exactly (these) two double quotes.
- "`%DOTDOT`" is the constant string "`..`".
- "`%NATNUMBER`" is a string of digits from 0 to 9, beginning with a digit from 1 to 9.
- "`%FLOATNUMBER`" is a string that either begins with a `%NATNUMBER`, followed by a dot "`.`", followed by a (possibly empty) string of digits from 0 to 9; or it begins with a dot and continues with a non-empty string of digits from 0 to 9.

One important difference to the definition of Table 4.3 is that the sequential composition of actions is here give in an "asymetrical" fashion: The action on the left side of a sequential composition must not be a sequential composition itself e.g., an action of the form "A_1 ; A_2 ; A_3 ; A_4" will be understood as "A_1 ; (A_2 ; (A_3 ; A_4))". This cannot cause any problems, because of the associativity of the sequential composition we proved in Proposition 4.5 on page 96.

Reserved keywords

As we already mentioned in Section 4.6.2, Atlantif identifiers must not be chosen among the keywords used in the syntax of Atlantif, nor among the keywords used in a model the Atlantif code is translated to.

For Atlantif itself, the following keywords are not usable:

```
among, and, any, bool, case, constant, dense, discrete, do, else, end,
external, false, for, from, function, hidden, if, in, infinity, init,
int, is, module, no, not, null, or, reset, return, select, silent, start,
stop, sync, then, time, to, true, type, unit, urgent, variables, visible,
wait, where, while
```

For the Uppaal translation (beyond those reserved by Atlantif), the following keywords are not usable:

```
chan, clock, commit, const, broadcast, process, state, guard, assign,
system, trans, deadlock, imply, forall, exists, return, typedef, struct,
rate, before_update, after_update, meta, priority, progress, scalar,
void, default, switch, continue, break
```

For the Tina translation (beyond those reserved by Atlantif), the following keywords are not usable:

```
net, pl, tr, pr, p, n, ne, t, c, w, e, h
```

For the data part of the Tina model, the reserved keywords of C also restrict types, functions and variables.

For the Fiacre translation (beyond those reserved by Atlantif), the following keywords are not usable:

`append`, `array`, `channel`, `component`, `const`, `dequeue`, `elsif`, `empty`, `enqueue`, `first`, `foreach`, `full`, `nat`, `none`, `of`, `out`, `par`, `port`, `priority`, `process`, `queue`, `read`, `record`, `states`, `union`, `var`, `write`

Appendix D

An extended summary in French

Un modèle intermédiaire pour la vérification des systèmes asynchrones embarqués en temps réel :

définition et application du langage ATLANTIF

D.1 Introduction

D.1.1 Motivation

Contexte général

Le monde de l'année 2009 dépend plus que jamais de systèmes informatiques, et cette dépendance se renforcera sans doute dans l'avenir. L'argent qui n'existe qu'électroniquement, l'élimination effective des distances grâce aux téléphones portables et à l'Internet ainsi que plusieurs autres manifestations de changements fondamentaux de notre culture sont aujourd'hui acceptés comme faisant partie de notre vie quotidienne.

Techniquement, beaucoup de ces changements ne sont devenus possible que grâce aux utilisations nombreuses de *systèmes embarqués*, c'est-à-dire de systèmes en informatique très spécialisés qui sont intégrés dans des équipements électriques ou électroniques et qui contrôlent ces équipements. Les tâches dont ils sont chargés sont normalement trop complexes pour être fait par des être humains (par exemple des calculs rapides de grands quantités de données), ou trop dangereuses (par exemple le contrôle d'un vaisseau spatial ou des manipulations dans des réacteurs nucléaires ou des usines chimiques), ou simplement trop ennuyeux (par exemple le contrôle de feux de circulation). Souvent, plusieurs systèmes embarqués (identiques ou différents) sont connectés et communiquent entre eux.

Evidement, ces dépendances nombreuses engendrent des risques et des vulnérabilités, puisque le malfonctionnement de systèmes critiques peut provoquer d'incidents facheux (comme des embouteillages) ou même des conséquences désastreuses (comme des catastrophes aériennes). Pour éviter cela, un développeur ne peut pas se fier uniquement à son intuition, parce que l'exécution en parallèle de plusieurs systèmes embarqués atteint facilement une complexité qui dépasse la portée de l'imagination humaine.

Alors il est vital de trouver une méthode systématique pour s'assurer que des systèmes informatiques fonctionnent correctement. Si on ne peut pas prendre le risque d'un malfonctionnement pendant l'utilisation, alors il est évident qu'il faut placer une telle méthode dans le développement de ces systèmes. Parmi les approches qui existent dans ce but il y a l'utilisation des méthodes formelles comme le *model checking*.

Le model checking commence par la description du comportement du système, couvrant des aspects comme les messages émis par le système (vers d'autres systèmes ou vers des utilisateurs humains), les messages que le système attend de recevoir d'autres systèmes, comment les données d'entrée sont procédées, combien de temps se passe avant l'action suivante, etc. L'écriture de cette description est faite par une notation standardisée, qui est soit purement textuelle, soit un mélange d'éléments textuels et graphiques, et qui est définie munie d'une sémantique formelle non ambiguë.

Lors de l'étape suivante, une ou plusieurs propriétés que le système doit satisfaire sont énoncées, par exemple « Il est impossible que le feu piétons A et le feu voitures B sont verts en même temps. », ou « Si la vitesse de l'avion est supérieur à 400 km/h, alors les volets ne sont pas sortis. », ou « Une lampe d'incident doit être allumée dans le cockpit de l'avion, si les dernières prévisions météo datent de plus de 30 minutes. ». Ces propriétés sont aussi exprimées dans un langage formel avec une sémantique non ambiguë.

Avec cette description formelle et une formule exprimant une propriété désirée, des algorithmes vérifient si le système satisfait la propriété. D'abord, ils génèrent un ensemble complet de toutes les configurations (« états ») qui peuvent être atteintes d'après la description du système, ils y inclus des informations sur quel état peut être succédé par quel autre état. Puis, toutes les successions d'états possibles sont vérifiées si elles satisfont la propriété ou non. Donc le model checking vérifie strictement et surtout exhaustivement si le système se comporte comme il le doit.

La modélisation abstraite

Données, concurrence et temps réel. Les langages textuelles et les modèles graphiques qu'on utilise pour les descriptions de système dans le model-checking doivent être assez simple pour permettre l'utilisation des algorithmes de vérification efficaces. Or, si l'objectif est l'application des méthodes formelles sur des systèmes *réalistes*, les langages simples ne sont pas aptes pour la modélisation de ces systèmes.

Dans cette thèse, le terme « réaliste » décrit trois aspects qui doivent être couvert par un langage adéquat : la gestion de données, la concurrence et le temps réel. Dans le reste de cette section, nous discuterons les concepts qui peuvent apparaître dans la modélisation des systèmes réalistes pour chacun de ces aspects.

Le premier aspect est la gestion de types de données complexes, couvrant les concepts suivants :

– *Représentation :* Types de données simples (tels que booléens, entiers et de types énumérés) et aussi des types structurés (tels que array, records, listes, unions, ensembles et arbres) peuvent tous les deux apparaître comme des paramètres dans des systèmes que nous souhaitons modéliser. Un langage adéquat doit donc contenir les moyens pour un utilisateur de définir des types.
  Par exemple, un système qui décrit un pylône relais GSM pourrait avoir un type *client* défini comme un array qui contient le numéro téléphone, le fournisseur d'accès etc., et un autre type *connections*, défini comme un ensemble de *clients*.
– *Manipulation :* Opérations et fonctions mathématiques doivent être représentées. Celles-ci peuvent être prédéfinies pour les types standards (ex. l'addition des entiers), mais clairement, les types définis par l'utilisateur n'ont de sens que si des fonctions peuvent être définies par l'utilisateur aussi.
  Par exemple, si on suppose une variable *clients_actuels* du type *connections*, des fonctions de mises à jour sont nécessaires pour représenter des nouveaux clients qui entrent ou des clients actuels qui sortent de la portée du pylône.

Le deuxième aspect couvre la *concurrence*, c'est-à-dire la description de systèmes qui sont composés de deux ou plusieurs sous-systèmes indépendants, que nous appellerons des *processus* dans la suite. Cet aspect couvre les concepts suivants :

– *Communication :* Si plusieurs processus sont exécutés en concurrence, il faut que la communication soit possible entre eux. Par exemple, dans un système qui décrit un avion, un processus *cockpit* envoie un message vers un processus *volets* pour ordonner leur sortie.
  Une communication entre des processus peut être soit *décalée* (un processus envoie un message, puis un ou plusieurs autres processus le reçoivent), soit *synchronisée* (tous les processus participants à la communication le font simultanément, donc il n'est pas nécessaire d'identifier un émetteur et des récepteurs).
– *Activation/Désactivation :* Pendant l'exécution d'un système, l'ensemble des processus participants n'est pas nécessairement statique : des nouveaux processus peuvent être créés, et des anciens processus peuvent disparaître. Par exemple, l'arrivée et le départ des nouveaux clients dans la portée du pylône pourraient être représentés par l'activation et la désactivation des différents processus, chacun correspondant à un client.

Le troisième aspect est la représentation du *temps*. Le modèle d'un processus exprime dans quel ordre ce processus fait ses communications. Or, parfois cette description *qualitative* n'est pas suffisante ; et il est alors nécessaire de fournir des informations *quantitatives* sur combien de temps se passe pendant l'exécution. Cet aspect couvre les concepts suivants :

– *Délais :* Un langage adéquat doit pouvoir exprimer l'inaction d'un processus pendant un certain temps. Normalement, c'est une représentation abstraite d'une activité qui nécessite du temps pour son exécution, comme un déplacement physique, par exemple la sortie des volets.
– *Urgence :* Si un processus est prêt pour exécuter une certaine communication, il peut avoir du sens de forcer l'exécution immédiate de cette communication, au lieu de laisser

le temps s'écouler.
Par exemple, si un téléphone portable reçoit un message par le processus pylône relais qui dit qu'il y a un appel arrivant, alors la sonnerie est lancée immédiatement.

- *Latence :* L'idée de l'urgence est complétée par celle de la latence. Celle-ci indique qu'une quantité limitée de temps peut passer avant qu'une communication devienne urgente.
  Pour rester dans l'exemple précédent, le processus du téléphone pourrait avoir besoin d'un temps indéfini mais très limité pour décider quelle sonnerie jouer.

Ces trois aspects ne sont pas entièrement indépendants : Des concepts supplémentaires doivent être considérés s'ils sont combinés, comme les suivantes :

- *Transmission des valeurs :* Dans un langage qui combine données et concurrence, il devrait être possible d'exprimer que les données générées dans un processus peuvent être utilisés dans un autre processus.
  Par exemple, le processus qui représente le capteur de vitesse d'un avion doit transmettre la valeur de la vitesse actuelle vers le processus qui représente l'autopilote.
- « *Timeout* » : Dans un langage qui combine temps et concurrence, les communications peuvent dépendre du passage du temps. Si le modèle permet à deux ou plusieurs processus de communiquer par synchronisation, alors le modèle devrait pouvoir exprimer qu'une fois que le processus est prêt à communiquer, il n'attend pas arbitrairement, mais il devient indisponible pour la communication après une certaine durée de temps.
  Par exemple, après que le pylône relais ait envoyé un message d'un appel arrivant vers un téléphone portable, il n'attend qu'une minute que le téléphone accepte l'appel.

Le besoin pour les modèles intermédiaires. La description et la vérification de systèmes couvrant les trois aspects des données, de la concurrence et du temps est un champ de recherche très active depuis déjà plus que deux décades. Fin des années 1980 et durant les années 1990, beaucoup des approches étaient dans une de deux catégories suivantes :

- Des *langages de haut-niveau* ont une syntaxe purement textuelle et fortement expressive qui permet des descriptions très concises. En particulier, des *algèbres de processus* [97], déjà capable de représenter concisément des données complexes et de la concurrence, ont été élargies avec de constructions de temps réel. Des algèbres de processus décrivent le comportement d'un système par l'ordre des *événements de communications*, qui représentent comment le système est perçu par l'extérieur, donc le comportement interne est ignoré. Ces événements internes sont combinés avec des opérateurs mathématiques, qui expriment par exemple que deux faits se passent l'un après l'autre (composition séquentielle) ou que deux comportements s'exécutent indépendamment (composition parallèle).
- Des *modèles graphiques* combinent des descriptions textuelles avec un composant visuel, ce qui permet une application plus intuitive. En particulier, des modèles comme des réseaux des automates et des réseaux de Petri, qui sont déjà munis des représentations intuitives de la concurrence, ont été élargis avec des constructions pour données et temps. Les tels modèles les plus connus sont des *automates temporisés* [4] et des *réseaux*

de Petri temporisés [95].
Basés sur des règles de syntaxe et de sémantique plus simples que dans le cas des algèbres de processus, ces modèles graphiques ont permis le développement des algorithmes efficaces et des outils logiciels pour la simulation et la vérification formelle.

Les expériences réalisées avec ces deux approches dans la modélisation des systèmes avec des données, de la concurrence et du temps ont montré des points forts et des points faibles complémentaires :

– Des langages de haut-niveau sont très expressifs et concis, mais leur structure élaborée rend la création des outils de vérification difficile. Dans certains cas, même la définition formelle de leur sémantique est assez complexe.
– Des modèles graphiques sont les langages d'entrée pour des différents outils de vérification, mais ils sont inadaptés pour faire des descriptions concises : la représentation d'un système complexe peut rapidement devenir peu structurée et illisible à cause des objets qui se chevauchent inévitablement.

Pour dépasser ces problèmes, des *modèles intermédiaires* ont été développés pendant les dernières années, avec une conception inspirée des deux approches, et visant à combiner les différents points forts. Dans le cas idéal, il est alors possible de spécifier un système dans un langage de haut-niveau, ainsi de traduire la spécification dans un modèle intermédiaire, et finalement de traduire celui-ci vers un modèle graphique, où des outils peuvent être appliqués pour faire du model checking, de la simulation etc. La plus grande difficulté dans une telle chaîne de traduction est la préservation de la sémantique. L'utilisation des résultats obtenues au niveau du modèle graphique pour raisonner sur la spécification initiale n'est possible que si la sémantique de cette spécification initiale a été préservée.

Une proposition pour un nouveau format intermédiaire. Le but de cette thèse est l'introduction du nouveau modèle intermédiaire Atlantif (*Asynchronous Timed Language Amplifying* Ntif). Bien que plusieurs propositions pour des modèles intermédiaires ont été faits pendant les dernières années, ils leurs manquent tous des idées importantes pour la représentation des langages de haut-niveau récents. Certains sont définis seulement sémi-formellement, d'autres ne couvrent qu'un ou deux de nos trois aspects, d'autres utilisent peu de constructions de haut-niveau ou les appliquent avec une approche sémantique différente.

Notre objectif pour la définition d'Atlantif est de surmonter ces restrictions et de proposer des traductions vers des outils de vérification.

Structure de ce résumé

Chacune des sections de ce résumé correspond à un chapitre de la thèse. La section D.2 (qui correspond au chapitre 2) discute les notations mathématiques nécessaires pour la suite. La section D.3 (chapitre 3) présente et analyse des langages et de modèles existantes, avec un accent sur les choix qui peuvent être fait pendant la définition d'un nouveau modèle. La section D.4 (chapitre 4) définit formellement la syntaxe et la sémantique d'Atlantif.

La section D.5 (chapitre 5) illustre à partir de deux exemples comment ATLANTIF peut représenter des constructions qui sont typiques pour les langages de haut-niveau. La section D.6 (chapitre 6) décrit deux traductions de sous-ensembles d'ATLANTIF vers d'autres modèles : les automates temporisés de l'outil UPPAALet les réseaux de Petri temporisés de l'outil TINA. La section D.7 (chapitre 7) donne un exemple d'application d'ATLANTIF (un ascenseur). Enfin, la section D.8 (chapitre 8) conclue ce résumé.

D.2 Notations

Des différentes notations sur des ensembles, des fonctions et d'autres objets mathématiques qui sont appliqués dans la suite sont dans leur majorité des notations standards. Toutes les constructions introduites par nous sont listées dans les tables 2.1 à 2.5 sur les pages 10 à 14.

D.3 Aperçu et classification des méthodes formelles

Dans cette section, nous faisons une analyse des modèles et des langages formelles existantes, qui combinent des données, de la concurrence et du temps réel. Nous les groupons dans des modèles sémantiques, des modèles graphiques, des langages de haut-niveau et enfin des modèles intermédiaires. Pour conclure, nous donnons une liste des approches différentes qui sont observables dans tous ces modèles et langages

D.3.1 Le modèle sémantique : Le système à transitions temporisées

Les définitions sémantiques dans cette thèse sont basées sur le modèle du système à transitions temporisées, qui est un modèle simple et sur lequel se basent la majorité des modèles et des langages temporisés existantes. Nous présentons ce modèle par les définitions suivantes :

Définition D.1. *(i) : Un* système à transitions étiquettées (STE) *est un 4-uplet* $(\Sigma, A, \rightarrow, S_0)$*, défini comme suit :*

- Σ *est un ensemble (possiblement infini) des* états*, écrits* S, S', S_0, S_1*, etc.*
- $S_0 \in \Sigma$ *est appelé l'*état initial.
- A *est un ensemble (possiblement infini) des* étiquettes discrètes*, écrites* a, a', a_0, a_1*, etc.* A *contient un élément spécial* τ*, appelé l'étiquette* silencieuse.
- *L'ensemble des 3-uplets* $\rightarrow \subseteq (\Sigma \times A \times \Sigma)$ *est appelé la* rélation de transition*. Nous écrivons «* $S \xrightarrow{a} S'$ *» au lieu de «* $(S, a, S') \in \rightarrow$ *».*

(ii) : Un domaine de temps *est une structure* $(\mathbb{D}, 0, <, +)$ *qui satisfait les conditions suivantes :*

- $\mathbb{D}$ *est un ensemble fini ou infini. Nous écrivons* t, t', t_1, t_2*, etc. ses éléments.*

– $<$ *est un ordre total sur* $\mathbb{D}$.
– 0 *est l'élément minimal par rapport à l'ordre* $<$.
– *L'opération binaire* $+$ *est entièrement définie sur* $\mathbb{D}$ *; elle est associative et commutative, et* 0 *est son élément neutre.*
– $(\forall t_1, t_2 \in \mathbb{D})\ (t_1 < t_2 \Leftrightarrow (\exists t_3 \in \mathbb{D})\ t_3 \neq 0 \wedge t_1 + t_3 = t_2)$
– $(\forall t, t')\ 0 < t \Rightarrow t' < t + t'$

Nous disons souvent aussi « domaine de temps » pour désigner l'ensemble sous-jacent $\mathbb{D}$.

(iii) : Nous disons que le domaine de temps est dense *si* $\mathbb{D}$ *est dense par rapport à* $<$, *c'est-à-dire* $(\forall t_1, t_2 \in \mathbb{D})\ t_1 < t_2 \Rightarrow (\exists t_3 \in \mathbb{D})\ t_1 < t_3 < t_2$. *Nous disons que le domaine de temps est* discret *si* $\mathbb{D}$ *est discret par rapport à* $<$, *c'est-à-dire* $(\forall t_1 \in \mathbb{D})\ (\exists t_2 \in \mathbb{D})\ t_1 + t_2$ *est le plus petit élément supérieur à* t_1 *(donc tout élément a un successeur direct).*

Des exemples pour des domaines denses sont les rationels non négatives $\mathbb{Q}_{\geq 0}$ *et les réels non négatives* $\mathbb{R}_{\geq 0}$. *Un exemple pour un domaine discret sont les entiers naturels* $(\mathbb{N}, 0, <, +)$.

(iv) : Un système à transitions temporisées (STT) *est un 5-uplet* $(\Sigma, A, \mathbb{D}, \rightarrow, S_0)$, *définit comme suit :*

– Σ, S_0 *et* A *sont définis comme dans (i).*
– $\mathbb{D}$ *est l'ensemble sous-jacent d'un domaine de temps tel que* $A \cap \mathbb{D} = \varnothing$. *Nous écrivons* l, l', l_0, l_1, *etc. pour les éléments de* $A \cup (\mathbb{D} \setminus \{0\})$.
– *L'ensemble des 3-uplets* $\rightarrow \subseteq (\Sigma \times (A \cup (\mathbb{D} \setminus \{0\})) \times \Sigma)$ *est appelé la* rélation de transition. *Nous écrivons «* $S \xrightarrow{l} S'$ *» pour un* $(S, l, S') \in \rightarrow$. *Si* $l \in \mathbb{D}$, *nous appelons* (S, l, S') *une* transition temporisée *; si* $l \in A$ *nous l'appelons une* transition discrète.

Nous identifions trois propriétés intuitives qui sont généralement souhaitées dans des STEs, données par la définition suivante :

Définition D.2. *Un* STT $(\Sigma, L, \mathbb{D}, \rightarrow, S_0)$ *est appelé* bien-temporisé, *si et seulement si toutes les conditions suivantes sont satisfaites :*

1. Additivité de temps *: Deux transitions temporisées successives sont égales à leur somme. Formellement, pour tous* $S_1, S_2 \in \Sigma, t_1, t_2 \in (\mathbb{D} \setminus \{0\})$ *:*
$$S_1 \xrightarrow{t_1+t_2} S_2 \text{ ssi } (\exists S_3 \in \Sigma)\ S_1 \xrightarrow{t_1} S_3 \xrightarrow{t_2} S_2$$
2. Déterminisme de temps *: À partir d'un état donné, l'écoulement de temps ne peut pas conduire à deux états différents. Formellement, pour tous* $S_1, S_2, S_3 \in \Sigma, t \in (\mathbb{D} \setminus \{0\})$ *:*
$$\text{si } S_1 \xrightarrow{t} S_2 \text{ et } S_1 \xrightarrow{t} S_3, \text{ alors } S_2 = S_3$$
3. Progrès maximal des actions urgentes *: Supposons un ensemble* $U \subseteq A$ *des étiquettes appelées* urgentes. *Un état qui permet une transition discrète par une étiquette urgente ne doit pas permettre une transition temporisée. Formellement, pour tout* $S_1 \in \Sigma, l \in A, t \in (\mathbb{D} \setminus \{0\})$ *:*
$$\text{si } l \in U \text{ et } S_1 \xrightarrow{l}\ , \text{ alors } \neg \left(S_1 \xrightarrow{t} \right)$$

D.3.2 Les modèles graphiques

Des modèles graphiques sont des modèles qui utilisent une notation graphique finie pour représenter des comportements possiblement infinis. Dans cette thèse, deux modèles graphiques sont d'une grande importance, qui sont les automates temporisés et les réseaux de Petri temporisés. Dans la suite, nous présentons ces deux modèles informellement.

Les automates temporisés

Les réseaux des automates temporisés sont utilisés dans beaucoup des outils pour la simulation et la vérification formelle, tels que Uppaal [89], Red [123], Sgm [80], Cmc [88], Kronos [126] et Rabbit [27].

Chaque automate temporisé est défini par un ensemble d'états discrets, par un ensemble des transitions entre ces états et par un ensemble des variables spéciales, dites *horloges*. À tout instant, un automate peut soit faire passer le temps (ce qu'augmente le valeur de toutes les horloges), soit prendre une transition pour passer d'un état discret à l'autre. La deuxième possibilité peut être restreinte par une formule sur la transition qui exige certaines valeurs pour les horloges. Une transition peut remettre la valeur de certaines horloges à zéro.

Dans un réseau d'automates temporisés, plusieurs automates communiquent par les étiquettes sur leurs transitions.

Un exemple simple d'un réseau d'automates temporisés est présenté dans Fig. 3.4 sur page 27.

Les réseaux de Petri temporisés

Les réseaux de Petri temporisés sont aussi utilisés dans beaucoup des outils pour la simulation et la vérification formelle, tels que Tina [16], Roméo [65], CPN Tools [107] et Oris [113].

Le modèle du réseau de Petri y est élargi par des intervalles qui sont associées à chaque transition et qui définissent à quel ensemble d'instants après la sensibilisation de la transition elle peut être tirée.

Un exemple simple pour un réseau de Petri est donné dans Fig. 3.5 sur page 30.

D.3.3 Les langages de haut-niveau

Pour plusieurs raisons, une approche graphique n'est souvent pas suffisant pour la modélisation des systèmes réalistes. Par exemple, dans un modèle graphique, l'ensemble des processus est statique (aucun processus ne peut être créé ou terminé pendant l'exécution), la communication est limitée aux concepts simples et les structures des états et des transitions sont rigides et artificielles. Bref, la simplicité des constructions disponibles peut faire la modélisation des systèmes complexes une tâche incommode.

Pour pallier cet inconvénient, des *langages de haut-niveau* ont été développés. Ces langages sont purement textuels et ils offrent des constructions puissantes.

Parmi ces langages, les *algèbres de processus* [14] présentent un choix naturel : la sémantique d'une algèbre de processus est donnée par un ensemble de règles, chacune transformant un terme de syntaxe vers un autre, où toute transformation correspond à une transition dans un STT ou un STE. Ces règles sont souvent données dans le style simple des « structural operational semantics » d'après Plotkin [105].

La liste suivante décrit les langages qui ont été, directement ou indirectement, influencés la définition d'ATLANTIF :

- Le langage CCS [97] contient des constructions pour exprimer entre autre la concurrence, la composition séquentielle, et le choix. Des extensions temporelles [98, 125] ont proposés des opérateurs pour exprimer des délais et des limiteurs de vie.
- Pour le langage CSP [77], des extensions [108, 103] ont proposés des opérateurs temporelles semblables.
- Le langage LOTOS [81] est un standard ISO pour la description formelle des systèmes concurrentes et communicantes.
  Parmi les extensions temporelles, ET-LOTOS [91] introduit un opérateur de *capture de temps*, et RT-LOTOS [47] introduit un opérateur de *latence*.
- Des différentes idées pour des extensions de LOTOS ont convergé vers les langages E-LOTOS (*extended* LOTOS) [83] et son dialecte LOTOS NT (LOTOS *Nouvelle Technologie*) [115]. Ces extensions incluent une composition parallèle généralisée, une composition séquentielle symétrique, la gestion des exceptions, et aussi des constructions de temps réel.

D.3.4 Les modèles intermédiaires

Des limites dans les approches des modèles graphiques et des langages de haut-niveau ont été observées pendant des années, en particulier par rapport à la modélisation des grands systèmes pour les modèles graphiques et par rapport à la vérification automatique pour des langages de haut-niveau. Pour pallier cet écart entre modélisation et vérification, l'approche des modèles intermédiaires a été établi [37, 61, 21], dont les idées centrales sont les suivantes :

- Ils sont munis des constructions de haut-niveau, tel que des types de données définis par l'utilisateur, opérateurs de choix ou opérateurs de communication élaborés.
- Ils permettent la vérification formelle, soit directement, soit indirectement, par traduction vers un autre modèle.

La liste suivante donne des exemples pour les modèles intermédiaires :

- BIP [11] décrit un système par plusieurs processus séquentiels, qui sont liés par des *connecteurs* qui décrivent comment les processus communiquent. BIP utilise du temps discret.
- Dans MODEST [30], les processus séquentiels permettent aussi des constructions probabilistes.

– NTIF (*New Technology Intermediate Form*) [61] a été conçu pour représenter des processus séquentiels qui gèrent des structures de données complexes. Les actions dans un processus NTIF sont données par des *transitions à branchement multiples*, qui permettent une description très concise. NTIF est défini sans constructions pour exprimer la concurrence ou le temps-réel.
– FIACRE [17] est partiellement basé sur NTIF, mais contient des constructions pour exprimer la concurrence ou le temps-réel.

D.3.5 Résumé et observations

Nous pouvons observer par la comparaison des différents modèles et langages que beaucoup des approches différentes sont possibles pour exprimer des données, de la concurrence et du temps-réel. La liste suivante donne plusieurs de ces approches :

– Plusieurs modèles permettent à l'utilisateur de définir ses propres types de données et fonctions.
– La majorité des modèles est définie avec une communication par synchronisation.
– La synchronisation peut être définie d'une manière plutôt simple (par exemple, la synchronisation binaire, limité à deux processus) ou d'une manière plus complexe (choix entre plusieurs ensembles de plusieurs processus).
– Quelques modèles sont capable de démarrer et d'arrêter dynamiquement des processus.
– L'envoi des données entre différents processus peut se faire soit par des variables partagées, soit par des *offres*, c'est-à-dire des valeurs envoyées lors d'une synchronisation.
– Les limiteurs de vie des communications sont définis soit à l'intérieur des processus (donc lors d'une communication, chaque processus peut imposer des contraintes temporelles), soit sur le niveau de composition (donc une communication a une contrainte temporelle globale).
– Quand un limiteur de vie est arrivé à sa limite, deux approches sémantiques différentes peuvent s'appliquer : soit la communication devient impossible dès que le temps continue de s'écouler, soit l'écoulement de temps est bloqué jusqu'à la communication a lieu. Inspiré par [36], nous écrivons respectivement *limite faible* et *limite forte* pour ces approches.

D.4 La syntaxe et la sémantique d'ATLANTIF

D.4.1 La syntaxe d'ATLANTIF

La syntaxe d'ATLANTIF, présentée dans la table D.1 est décrit dans une variante d'EBNF (*Extended Backus-Naur Form* [82]) où les parties entre des « `[`, `]` » sont optionnelles et des barres verticales désignent des alternatives. ATLANTIF est un superensemble strict de NTIF ; des ombres en gris sont utilisés pour mieux montrer les extensions, que nous détaillerons dans les sections D.4.2 et D.4.3.

Pour simplifier, nous ne détaillerons pas des définitions de types (qui inclurent de types

Syntaxe de modules :	
X ::= **module** M **is**	
[(**no** | **discrete** | **dense**) **time**]	(*option temporelle*)
type T_1 **is** $D_1 \ldots$ **type** T_n **is** D_n	(*déclarations de types*)
function F_1 **is** $Y_1 \ldots$ **function** F_k **is** Y_k	(*déclarations de fonctions*)
$R_1 \ \ldots \ R_m$	(*synchroniseurs, définis en bas*)
init $u_0, \ldots, u_j$	(*unités initialement actives*)
$U_0 \ \ldots \ U_l$	(*définitions d'unités, définis en bas*)
end module	
Syntaxe d'unités :	
U ::= **unit** u **is**	
[**variables** V_0 `:`T_0 [`:=` E_0], ..., V_n `:`T_n [`:=`E_n]]	(*variables locales*)
from s_0 A_0 ... **from** s_m A_m	(*liste de transitions*)
$U_1 \ \ldots \ U_l$	(*sous − unités*)
end unit	
Syntaxe d'actions :	
A ::= $V_0, \ldots, V_n$ `:=` $E_0, \ldots, E_n$	(*affectation déterministe*)
| $V_0, \ldots, V_n$ `:=` **any** $T_0, \ldots, T_n$ [**where** E]	(*affectation non−déterministe*)
| **reset** $V_0, \ldots, V_n$	(*reset des variables*)
| **wait** E	(*delai*)
| G $O_1 \ldots O_n$ [[**must** | **may**] **in** W]	(*communication par porte*)
| **to** s	(*saut vers état*)
| **stop**	(*arrêter unité*)
| A_1 `;` A_2	(*composition sequentielle*)
| **if** E **then** A_1 **else** A_2 **end** [**if**]	(*conditionel*)
| **case** E **is** P_0`->`A_0 `|` ... `|` P_n`->`A_n **end** [**case**]	(*choix déterministe*)
| **select** A_0 `[]` ... `[]` A_n **end** [**select**]	(*choix non−déterministe*)
| **while** E **do** A_0 **end** [**while**]	(*boucle*)
| **null**	(*inaction*)
Syntaxe d'offres :	
O ::= `!`E	(*émission devaleur*)
| `?`P	(*réception devaleur*)
Syntaxe d'expressions :	
E ::= V	(*variable*)
| F`(`$E_1, \ldots, E_n$`)`	(*fonction*)
| C`(`$E_1, \ldots, E_n$`)`	(*constructeur*)
Syntaxe de motifs :	
P ::= **any** T	(*variable anonyme*)
| P_0 **where** E	(*condition*)
| `(`P_0`)`	
| V	(*variable*)
| C`(`$P_1, \ldots, P_n$`)`	(*constructeurr*)
Syntaxe de fenêtres temporels :	
W ::= `[`E_1`,`E_2`]` | `]`E_1`,`E_2`]` | `[`E_1`,`E_2`[` | `]`E_1`,`E_2`[`	(*intervalle borné*)
| `[`E_1`, ...[` | `]`E_1`, ...[`	(*intervalle non borné*)
| W_1 **or** W_2 | W_1 **and** W_2 | `(`W_0`)`	(*intervalles combinés*)
Syntaxe de synchroniseurs :	
R ::= **sync** G [`:` B] **is** K	(*formule de synchronisation*)
[**stop** $u_1, \ldots, u_m$] [**start** $u'_1, \ldots, u'_n$]	(*unités arrêtées et démarrées*)
end sync	
Syntaxe auxiliaire de synchroniseurs :	
K ::= u	(*unité seule*)
| K_1 **and** K_2	(*synchronisation*)
| K_1 **or** K_2	(*alternative*)
| N **among** `(`$K_1, \ldots, K_m$`)`	
| `(`K_0`)`	
N ::= n	(*entier naturel*)
| N_1 **or** N_2	(*choix*)
B ::= **visible** | **hidden** | **urgent** | **silent**	

TAB. D.1 – Syntaxe intégrale d'ATLANTIF

de données complexes, comme des records, des listes, etc.) et des définitions de fonctions.

D.4.2 Les processus séquentiels en ATLANTIF

Un processus séquentiel en ATLANTIF, appelé une unité, contient des déclarations de variables et optionnellement des *sous-unités*. Nous écrivons $decl(u)$ pour l'ensemble de variables déclarées dans une unité donnée u. Celles-ci peuvent être lues et/ou écrites dans les sous-unités de u, ce que permet de partager ces variables entre des différentes unités. Pour cela, chaque variable V est munie d'une portée, donnée par l'ensemble $accessible(V)$, qui définit les unités dans lesquelles V peut être lue ou écrite.

Chaque unité contient une liste des états discrets, le premier duquel est compris d'être l'état initial. À chaque état discret s nous attribuons une transition *à branchement multiple* avec la forme « **from** s A », où A est une action, notée act. En différence des modèles habituels, où les actions sont simplement des couples « condition/affectation », les actions d'ATLANTIF utilisent des constructions de haut-niveau, qui combinent des actions atomiques. Une action particulière est la *communication par une porte*, qui permet l'échange de données sous forme d'offres, dont chacune est soit de la forme $!E$ (représentant une émission du valeur de l'expression E), soit de la forme $?P$ (représentant une réception d'un valeur dans un motif P par *pattern-matching*).

En ce qui concerne le temps réel, ATLANTIF permet le temps discret (qui correspond à un domaine de temps isomorphe à $\mathbb{N}$) aussi que le temps dense (qui correspond à $\mathbb{R}_{\geq 0}$) ; le comportement sans temps quantitative est également permis. Cette option temporelle est donnée dans le header d'une spécification (par les mots-clés « **discrete time** », « **dense time** » ou « **no time** » ; ce dernier étant le défaut si l'option n'est pas spécifie). ATLANTIF a une action « **wait** » qui permet une quantité donné de temps à s'écouler (ce concept est emprunté des algèbres de processus tels que TCSP [108]), et les extensions suivantes à la communication par une porte :

- *Une fenêtre de temps* W composée des intersections (« **and** ») et unions (« **or** ») des intervalles ouvertes ou fermées, où « ... » représente l'infini. La communication peut être exécutée quand le temps écoulé depuis l'arrivée à l'action de communication est contenu dans la fenêtre de temps. Si W n'est pas spécifiée, elle est « $[0, ...[$ » par défaut. La fenêtre de temps prend donc le rôle d'un « *limiteur de vie* », tel qu'il se trouve dans des différents algèbres de processus temporisés tels que ET-LOTOS [91].
- *Une modalité* Q parmi **must** ou **may**, où **must** indique que la communication doit avoir lieu avant la fin de la fenêtre de temps (qu'on appelle l'« *échéance* »), et **may** indique que le temps peut s'écouler indéfiniment. Si Q n'est pas spécifié, elle est **may** par défaut. Le **may** représente donc un limiteur de vie avec une limite *faible*, pendant que le **must** represente un limiteur de vie avec une limite *forte*. Les réseaux de Petri temporisés et FIACRE ne permettent que des limites fortes, pendant que les automates temporisés et la plupart des extensions de LOTOS permettent une combinaison des limites fortes et faibles, ce qui justifie notre choix en ATLANTIF.

Sémantique

Sémantique statique. En ce qui concerne la sémantique statique, ATLANTIF hérite les règles de NTIF [61] avec des extensions pour les nouveaux constructions de syntaxe. Les règles héritées concernent le bon typage et la restriction à au plus une communication sur chaque chemin possible dans une transition à branchement multiple. L'extension des règles concerne l'initialisation correcte de variables avant leur utilisation, qui doit maintenant prendre en compte la concurrence et le partage des variables entre les unités.

Nous ajoutons les contraintes qu'une action « **wait** » n'est pas admise après une action de communication sur un chemin d'une transition à branchement multiple, que la fenêtre de temps de chaque communication « **must** » est soit non bornée, soit close à droite, que les communications par des synchroniseurs silencieux (« **silent** ») ne sont pas limités par une fenêtre de temps (c'est-à-dire, ces communications sont exécutées au plus tôt), et qu'une unité ne peut pas être active si une de ces sous-unités (directes ou indirectes) est active. En plus, aucune unité qui est permise de lire une variable V ne peut être active en même temps qu'une autre unité qui est permise de lire et/ou écrire V. Nous avons définis des algorithmes qui peuvent vérifier ce dernier critère.

Sémantique dynamique – définitions. Pour présenter la sémantique dynamique, nous avons d'abord besoin des définitions suivantes, héritées de NTIF. Nous supposons un ensemble *Val* de *valeurs*, dont les éléments sont écrits v, v', v_0, v_1, etc. Nous écrivons V pour l'ensemble de variables. Des fonctions partielles sur $\mathcal{V} \to \mathit{Val}$, appelées *environnements*, sont écrites $\rho, \rho', \rho_0, \rho_1$, etc. Nous écrivons $\mathit{dom}(\rho)$ pour le domaine de ρ. Les opérateurs *update* (*mis à jour*) $\oslash$ et *restriction* $\ominus$ sont définis sur les environnements de la manière suivante :

$$\begin{array}{l}
\rho \oslash \rho' \stackrel{\text{def}}{=} \rho'' \text{ où } \rho''(V) = \text{si } V \in \mathit{dom}(\rho') \text{ alors } \rho'(V) \text{ sinon } \rho(V) \\
\rho \ominus \{V_1, \dots, V_n\} \stackrel{\text{def}}{=} \rho'' \text{ où } \mathit{dom}(\rho'') = \mathit{dom}(\rho) \setminus \{V_1, \dots, V_n\} \\
\qquad\qquad\qquad\qquad\qquad \text{et } (\forall V \in \mathit{dom}(\rho'')) \ \rho''(V) = \rho(V)
\end{array}$$

La sémantique des expressions est donnée par un prédicat $\mathit{eval}(E, \rho, v)$ qui est **vrai** si et seulement si l'évaluation de l'expression E avec un environnement ρ donne un valeur v. La sémantique des motifs est donnée par une fonction du « *pattern-matching* » $\mathit{match}(v, \rho, P)$, qui renvoie « **fail** », si v ne peut pas être évaluée par ρ ; sinon elle renvoie un nouveau environnement ρ', qui correspond à ρ où les variables de P sont affectés avec les sous-termes applicables de v. La sémantique des offres est donnée par une fonction $\mathit{accept}(v, \rho, O)$, définie par :

$$\begin{array}{l}
\mathit{accept}(v, \rho, !E) \stackrel{\text{def}}{=} \text{si } \mathit{eval}(E, \rho, v) \text{ alors } \rho \text{ sinon } \mathbf{fail} \\
\mathit{accept}(v, \rho, ?P) \stackrel{\text{def}}{=} \mathit{match}(v, \rho, P)
\end{array}$$

Nous écrivons $\mathcal{S}$ pour l'ensemble d'identificateurs d'états munis de deux éléments spéciaux δ et Ω, réservés pour la sémantique. δ représente un état discret auxiliaire qui dénote la terminaison d'une action, ce que permet l'exécution des actions consécutives ; Ω représente la terminaison d'une unité.

Nous avons aussi besoin des définitions suivantes. Nous écrivons $\mathbb{D}$ le domaine de temps, t, t', t_0, t_1, etc. ses éléments, et $\mathbb{L}_1 \overset{\text{def}}{=} \{G\ v_1 \ldots v_n \mid G \in \mathbb{G},\ v_1, \ldots, v_n \in \mathit{Val}\} \cup \{\varepsilon\}$ l'ensemble d'étiquettes, où $\mathbb{G}$ est l'ensemble de portes et ε représente des transitions sans action de communication. L'opérateur binaire « + » est partiellement défini sur $\mathbb{L}_1 \times \mathbb{L}_1 \to \mathbb{L}_1$ par $l + \varepsilon \overset{\text{def}}{=} l$, $\varepsilon + l \overset{\text{def}}{=} l$, il est indéfini si les deux arguments sont différents de ε. Nous écrivons $\mathbb{U}$ pour l'ensemble d'identificateurs d'unités et $\mathcal{U}, \mathcal{U}', \mathcal{U}_0, \mathcal{U}_1$, pour ses sous-unités. Nous utilisons la notation $\rho_{\restriction \mathcal{U}}$ pour restreindre le domaine d'un environnement ρ aux variables accessibles dans un ensemble d'unités $\mathcal{U}$; formellement $\rho_{\restriction \mathcal{U}} \overset{\text{def}}{=} \rho \ominus \{V \mid (\forall u \in \mathcal{U})\ u \notin \mathit{accessible}(V)\}$. La sémantique des fenêtres de temps est définie par un prédicat $\mathit{win\_eval}(W, \rho, D)$ qui est **vrai** si et seulement si l'évaluation de W avec un environnement ρ génère un ensemble (possiblement infini) de temps D. Nous définissons également une fonction booléenne $\mathit{up\_lim}(Q, W, \rho, t)$ renvoyant **vrai** si et seulement si $Q = \textbf{must}$ et que l'ensemble D défini par $\mathit{win\_eval}(W, \rho, D)$ ait un maximum qui vaut t.

Sémantique dynamique – constructions séquentielles. Dans Ntif, la sémantique des actions a été défini par une relation ayant la forme $(A, \rho) \overset{l}{\Longrightarrow} (s, \rho')$, où A est une action, ρ, ρ' sont des environnements, $s \in \mathcal{S}$ est un état discret et $l \in \mathbb{L}_1$ est une étiquette [61].

Dans Atlantif, cette relation est élargit vers la forme $(A, d, \rho) \overset{l}{\Longrightarrow} (s, d', \rho')$, où d, d' ont la forme (t, μ), avec $t \in \mathbb{D}$ est appelé une *phase* (intuitivement, t représente le temps qui peut s'écouler dans l'unité avant la communication suivante), et μ une valeur booléen (appelé la *condition de blocage*), qui vaut **vrai** si et seulement si le temps ne peut pas s'écouler au-delà de t. Donc le prédicat signifie que l'action A dans le contexte de d et de ρ évolue vers un état local (s, d', ρ') (états locaux sont aussi écrits $\sigma, \sigma', \sigma_0, \sigma_1$, etc.), en produisant une transition étiquettée par l. Ces règles sont détaillées dans la figure D.1, où une trame grise indique une extension par rapport à Ntif.

La figure D.2 présente un exemple d'un système composé d'un utilisateur et d'une lampe. L'utilisateur, modélisé par l'unité *User*, appuie de façon répétée sur un interrupteur (représenté par une porte « *Push* »). La lampe, modélisée par l'unité *Lamp*, met à disposition trois niveaux de luminosité, qui sont modélisés par les trois états discrets « *Off* », « *Low* » et « *Bright* ». Quand la lampe est éteinte (état « *Off* »), appuyer sur l'interrupteur l'allume avec une luminosité basse (état « *Low* »). Si l'appui suivant est fait dans les cinq secondes suivantes, alors la lampe devient plus lumineuse (état « *Bright* »). Si c'est après plus que cinq secondes, alors la lampe est éteinte.

D.4.3 Concurrence dans ATLANTIF

Dans Atlantif, une spécification contient plusieurs unités qui sont synchronisées au respect des *synchroniseurs* (voir figure D.1), qui sont une généralisation des vecteurs de

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

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

$$(reset)\ \frac{}{(\mathbf{reset}\ V_0,\ldots,V_n,d,\rho)\overset{\varepsilon}{\Longrightarrow}(\delta,d,\rho\ominus\{V_0,\ldots,V_n\})}$$

$$(wait)\ \frac{eval(E,\rho,v)\wedge t\geq v\geq 0}{(\mathbf{wait}\ E,(t,\mu),\rho)\overset{\varepsilon}{\Longrightarrow}(\delta,(t-v,\mu),\rho)}$$

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

$$(to)\ \frac{}{(\mathbf{to}\ s,d,\rho)\overset{\varepsilon}{\Longrightarrow}(s,d,\rho)}\qquad (stop)\ \frac{}{(\mathbf{stop},d,\rho)\overset{\varepsilon}{\Longrightarrow}(\Omega,d,\rho)}$$

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

$$(case)\ \frac{eval(E,\rho,v)\wedge(\forall j<k)\ match(v,\rho,P_j)=\mathbf{fail}\wedge match(v,\rho,P_k)=\rho_k\neq\mathbf{fail}\wedge(A_k,d,\rho_k)\overset{l}{\Longrightarrow}\sigma}{(\mathbf{case}\ E\ \mathbf{is}\ P_0\mathtt{->}A_0\,\mathtt{|}\,\ldots\,\mathtt{|}\,P_n\mathtt{->}A_n\ \mathbf{end},d,\rho)\overset{l}{\Longrightarrow}\sigma}$$

$$(if_1)\frac{eval(E,\rho,\mathbf{true})\wedge(A_1,d,\rho)\overset{l}{\Longrightarrow}(s,d',\rho')}{(\mathbf{if}\ E\ \mathbf{then}\ A_1\ \mathbf{else}\ A_2\ \mathbf{end},d,\rho)\overset{l}{\Longrightarrow}(s,d',\rho')}\ (if_2)\frac{eval(E,\rho,\mathbf{false})\wedge(A_2,d,\rho)\overset{l}{\Longrightarrow}(s,d',\rho')}{(\mathbf{if}\ E\ \mathbf{then}\ A_1\ \mathbf{else}\ A_2\ \mathbf{end},d,\rho)\overset{l}{\Longrightarrow}(s,d',\rho')}$$

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

$$(while_1)\ \frac{eval(E,\rho,\mathbf{true})\wedge(A_0\ \mathbf{;while}\ E\ \mathbf{do}\ A_0\ \mathbf{end},d,\rho)\overset{l}{\Longrightarrow}\sigma}{(\mathbf{while}\ E\ \mathbf{do}\ A_0\ \mathbf{end},d,\rho)\overset{l}{\Longrightarrow}\sigma}$$

$$(while_2)\ \frac{eval(E,\rho,\mathbf{false})}{(\mathbf{while}\ E\ \mathbf{do}\ A_0\ \mathbf{end},d,\rho)\overset{\varepsilon}{\Longrightarrow}(\delta,d,\rho)}\qquad (null)\ \frac{}{(\mathbf{null},d,\rho)\overset{\varepsilon}{\Longrightarrow}(\delta,d,\rho)}$$

FIG. D.1 – Règles pour la sémantique dynamique des unités

```
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
```

FIG. D.2 – Programme ATLANTIF qui décrit un interrupteur

synchronisation [6, 33]. Un synchroniseur est activé chaque fois qu'une unité arrive à une action de communication, c'est-à-dire chaque fois qu'elle veut proposer un rendez-vous à son environnement. Il décrit comment des unités synchronisent et il détermine l'ensemble des unités qui tournent (qui sont *actives*). Formellement, un synchroniseur est de la forme « **sync** $G : B$ **is** K **stop** $u_1, \ldots, u_m$ **start** $u'_1, \ldots, u'_n$ **end sync** », où :

- G est une porte qui déclenche le synchroniseur.
- B est une balise facultative (noté $tag(G)$), qui peut prendre une des quatre formes suivantes : **visible** engendre une transition étiquetée par G et les offres échangées par G ; **hidden** engendre une transition interne qu'on appelle τ-transition ; **urgent** se comporte comme le dernier, mais en plus, le temps est bloqué si une synchronisation est possible ; et **silent** indique que la communication n'engendre pas une transition. Si aucune balise n'est spécifiée, alors le synchroniseur est **visible**.
- K est une formule composée d'identificateurs d'unités et d'opérateurs booléennes, qui décrit des combinaisons des unités qui doivent se synchroniser ; chacune de ces combinaisons est appelée un « *ensemble de synchronisation* ». L'ensemble d'ensembles de synchronisation attaché à G, noté $sync(G)$, est défini comme suit :
$$
\begin{aligned}
&sync(u) = \{\{u\}\}\\
&sync(K_1 \textbf{ and } K_2) = \{S_1 \cup S_2 \mid S_1 \in sync(K_1) \wedge S_2 \in sync(K_2)\}\\
&sync(K_1 \textbf{ or } K_2) = sync(K_1) \cup sync(K_2)\\
&sync(n \textbf{ among } (K_1, \ldots, K_m)) = sync(K'_1 \textbf{ or} \ldots \textbf{or } K'_k), \text{ où}\\
&\quad \{K'_1, \ldots, K'_k\} = \{(K_{i_1} \textbf{ and} \ldots \textbf{and } K_{i_n}) \mid 1 \leq i_1 < \ldots < i_n \leq m\}\\
&sync(n_1 \textbf{ or} \ldots \textbf{or } n_l \textbf{ among } (K_1, \ldots, K_m)) =\\
&\quad sync(n_1 \textbf{ among } (K_1, \ldots, K_m) \textbf{ or} \ldots \textbf{or } n_l \textbf{ among } (K_1, \ldots, K_m))
\end{aligned}
$$
- « **stop** $u_1, \ldots, u_m$ » et « **start** $u'_1, \ldots, u'_n$ » sont des constructions facultatives qui indiquent que les unités $u_1, \ldots, u_m$ deviennent inactives, pendant que $u'_1, \ldots, u'_n$ deviennent actives quand le synchroniseur est déclenché. Nous écrivons $stop(G) = \{u_1, \ldots, u_m\}$ et $start(G) = \{u'_1, \ldots, u'_n\}$. Par défaut, $stop(G) = \varnothing$ et $start(G) = \varnothing$.

Pour exprimer la concurrence, dans d'autres modèles intermédiaires (tels que les réseaux CÆSAR [59] ou les « *communicating state machines* » [84]) les transitions sont composés des communications de plusieurs processus (similaire aux transitions des réseaux de Petri).

Un inconvénient de cette approche est que le nombre des transitions dans le modèle résultat peut être le produit du nombre de transitions dans chaque processus. Des synchroniseurs offrent une approche plus symbolique pour éviter ce problème, mais cette approche est en même temps assez générale pour exprimer les concepts suivants :

- La compétition entre des processus en synchronisation peut être exprimée par des synchroniseurs qui décrivent plusieurs ensembles de synchronisation. Par exemple, dans la formule « u_1 **and** (u_2 **or** u_3) », u_2 et u_3 sont en compétition pour se synchroniser avec u_1.
- La synchronisation « *multiway* » (c'est-à-dire entre un nombre arbitraire de processus) peut être exprimée par des ensembles de synchronisation qui contiennent plus que deux unités. Par exemple, dans la formule « u_1 **and** u_2 **and** u_3 », les trois unités u_1, u_2 et u_3 doivent se synchroniser toutes ensemble.
- L'opérateur généralisé de la composition parallèle de [64] peut aussi être exprimé. Par exemple, la formule « **par** $G\#2, G\#3$ **in** $u_1||u_2||u_3$ **end par** » (qui signifie que deux ou trois processus parmi u_1, u_2 et u_3 peuvent se synchroniser sur G) peut être exprimée par « **sync** G **is** 2 **or** 3 **among** (u_1, u_2, u_3) **end sync** ».
- Des processus peuvent être démarrés ou arrêtés par eux-mêmes ou par des processus concurrentes. Par exemple, « **sync** G **is** u_1 **and** u_2 **stop** u_1, u_2 **start** u_3, u_4 **end sync** » signifie que les unités u_1 et u_2 sont arrêtées dès leur synchronisation sur G, et que u_3 et u_4 sont démarrées le même instant.

Sémantique dynamique – constructions séquentielles. À la différence de Ntif, qui n'a pas une sémantique parallèle, Atlantif est muni d'une deuxième couche de la sémantique pour la concurrence et le temps réel. Elle est donnée par un Stt (voir la définition D.1) avec la forme $(\mathbb{S}, \mathbb{T}, S_0)$ où :

- $\mathbb{S}$ est un ensemble d'*états globaux* (ou simplement *états*) de la forme (π, θ, ρ) (notés S, S', S_0, S_1, etc.), où $\pi : \mathbb{U} \to \mathcal{S}$ est une fonction partielle, appelée la *distribution d'états*, qui associe toute unité active à son état discret actuel, $\theta : \mathbb{U} \to (\mathbb{D} \times \mathbf{Bool})$ est une fonction partielle, appelée la *distribution de temps*, qui associe toute unité active à sa phase et sa condition de blocage actuelle, et ρ est un environnement. À remarque que l'ensemble des unités actives est donné par $dom(\pi)$ et $dom(\theta)$, avec $dom(\pi) = dom(\theta)$.
- $\mathbb{T}$ est un ensemble de transitions défini par une relation en $\mathbb{S} \times \mathbb{L}_2 \times \mathbb{S}$, où $\mathbb{L}_2 \stackrel{\text{def}}{=} (\mathbb{L}_1 \setminus \{\varepsilon\}) \cup \{\tau\} \cup (\mathbb{D} \setminus \{0\})$. Les transitions étiquetées dans $\mathbb{D} \setminus \{0\}$ sont les transitions de temps, les autres sont les transitions discrètes.
- $S_0 \in \mathbb{S}$ est l'état initial, qui est défini par $S_0 \stackrel{\text{def}}{=} (\pi_{0|\mathcal{U}_0}, \theta_{0|\mathcal{U}_0}, \rho_{0|\mathcal{U}_0})$, où π_0 est une fonction qui associe toute unité à son état discret initial (défini implicitement comme le premier état discret dans cette unité), θ_0 est la fonction qui renvoie constamment $(0, \mathbf{false})$ pour toute unité, ρ_0 est l'environnement qui associe toute variable à sa valeur initiale (si définie), et $\mathcal{U}_0$ est l'ensemble d'unités initialement actives. $\pi_{0|\mathcal{U}_0}$ et $\theta_{0|\mathcal{U}_0}$ représentent respectivement π_0 et θ_0, où le domaine est restreint à $\mathcal{U}_0$.

Une transition discrète correspond à une chaîne de zéro ou plus synchronisations silencieuses suivies par une synchronisation non silencieuse (appelé *chaîne* dans la suite). Nous appelons une *chaîne incomplète* toute préfixe d'une chaîne. Une chaîne est exécutée sans écoulement de temps.

Comme déjà observé en [59], l'exploration de toutes les chaînes possibles ne serait pas correct. Au lieu de cela, une synchronisation silencieuse n'est permis dans une chaîne que si au moins une des unités synchronisées et non arrêtées *ou* une des unités démarrées se synchronisent aussi dans une autre synchronisation dans la suite de la chaîne. Nous appelons l'*affectation d'unités* d'une synchronisation l'ensemble contenant les unités qui se synchronisent mais qui ne sont pas arrêtées et les unités qui sont démarrées. Nous associons à toute chaîne incomplète l'ensemble α d'affectations d'unités qui correspondent à celles synchronisations dans la chaîne incomplète, dont aucune unité ne s'est synchronisée dans la suite de la chaîne incomplète. Alors, il ne faut explorer que les chaînes qui finissent avec un ensemble α vide. Nous définissons les prédicats suivants :

– Le prédicat $\mathit{synchronizing}((S,\alpha),l,\mu,(S',\alpha'))$, défini sur $(\mathbb{S} \times \mathcal{P}(\mathcal{P}(\mathbb{U}))) \times (\mathbb{L}_1 \setminus \{\varepsilon\}) \times \mathbf{Bool} \times (\mathbb{S} \times \mathcal{P}(\mathcal{P}(\mathbb{U})))$ est **vrai** si et seulement si (1) une transition étiquetée par l peut avoir l'origine dans l'état global S et aller vers l'état global S', (2) la disjonction des conditions de blocage dans les états locaux atteintes par cette transition vaut μ et (3) un ensemble d'affectations d'unités α évolue vers α' par cette synchronisation. Formellement :

$$\begin{array}{l}
\mathit{synchronizing}(((\pi,\theta,\rho),\alpha), G\ v_1 \dots v_n, \mu, ((\pi',\theta',\rho'),\alpha')) \stackrel{\text{def}}{=} \\
\quad (\exists\{u_1,\dots,u_m\} \in \mathit{sync}(G))\ \{u_1,\dots,u_m\} \subseteq \mathit{dom}(\pi) \land \\
\qquad (\forall i \in 1..m)\ ((\mathit{act}(\pi(u_i)),\theta(u_i),\rho_{\lceil\{u_i\}})\ {}^{G}\!\xrightarrow{v_1 \dots v_n} (s_i,(t_i,\mu_i),\rho_i) \land s_i \neq \delta) \land \\
\qquad \mu = \bigvee_{i=1..m} \mu_i \land \\
\qquad \mathit{next_\pi}(\pi, [u_i \mapsto s_i \mid i \in 1..m], G, \pi') \land \\
\qquad \mathit{next_\theta}(\theta, \{u_1,\dots,u_m\}, \mathit{min}_{i \in 1..m}(t_i), G, \theta') \land \\
\qquad \mathit{next_\rho}(\rho, [V \mapsto \rho_i(V) \mid u_i \in \mathit{accessible}(V)], \mathit{dom}(\pi'), G, \rho') \land \\
\qquad \mathit{next_\alpha}(\alpha, \{u_1,\dots,u_m\}, G, \alpha')
\end{array}$$

Le prédicat $\mathit{next\_\pi}$ définit la nouvelle distribution d'états après une synchronisation.

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

Le prédicat $\mathit{next\_\theta}$ définit la nouvelle distribution de temps après une synchronisation. Si la synchronisation a été silencieuse, alors la nouvelle phase t_0 des unités affectées est donnée par la phase minimale parmi les unités qui se synchronisent. Donc elle correspond à l'unité / aux unités dans laquelle / lesquelles le délai le plus longue a eu lieu, c'est-à-dire l'unité laquelle les autre unités qui se synchronisent ont du attendre. Si la synchronisation n'a pas été silencieuse, la nouvelle phase de toutes les unités affectées est mis à zéro.

$$\mathit{next\_\theta}(\theta,\mathcal{U},t_0,G,\theta') \stackrel{\text{def}}{=}$$

$$\theta' = \begin{cases} ((\theta \oslash [u \mapsto (t_0,\mathbf{false}) \mid u \in \mathcal{U}])) \ominus \mathit{stop}(G) & \\ \quad \oslash [u \mapsto (t_0,\mathbf{false}) \mid u \in \mathit{start}(G)] & \text{si } \mathit{tag}(G) = \textbf{silent} \\ ((\theta \oslash [u \mapsto (0,\mathbf{false}) \mid u \in \mathcal{U}]) \ominus \mathit{stop}(G)) & \\ \quad \oslash [u \mapsto (0,\mathbf{false}) \mid u \in \mathit{start}(G)] & \text{sinon} \end{cases}$$

Le prédicat $\mathit{next\_\rho}$ définit le nouveau environnement après une synchronisation. Le domaine de l'environnement est restreinte par rapport au nouveau ensemble d'unités actives. Pour toute unité démarrée qui déclare une variable V avec un valeur initial, $[V \mapsto \rho_0(V)]$ est ajouté à l'environnement.

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

Le prédicat *next_*α, qui définit les nouvelles affectations d'unités après une synchronisation, commence par supprimer les affectations d'unités dont au moins une unité s'est synchronisée. Si la synchronisation a été silencieuse, un nouvel ensemble est ajouté, qui contient toutes les unités synchronisées et non arrêtées et toutes les unités démarrées.

$$
\textit{next\_}\alpha(\alpha, \mathcal{U}, G, \alpha') \stackrel{\text{def}}{=}
$$

$$
\alpha' = \begin{cases} (\alpha \setminus \{\mathcal{U}' \in \alpha \mid (\exists u \in \mathcal{U})\ u \in \mathcal{U}'\}) \cup & \\ \qquad \{(\mathcal{U} \setminus \textit{stop}(G)) \cup \textit{start}(G)\} & \text{si } \textit{tag}(G) = \textbf{silent} \\ \alpha \setminus \{\mathcal{U}' \in \alpha \mid (\exists u \in \mathcal{U})\ u \in \mathcal{U}'\} & \text{sinon} \end{cases}
$$

où les opérateurs $\oslash$, $\ominus$ sont définies sur π et sur θ d'une manière pareille que sur ρ.

– Le prédicat $\textit{enabled}(S, l, \mu, S')$, définit sur $\mathbb{S} \times (\mathbb{L}_1 \setminus \{\varepsilon\}) \times \textbf{Bool} \times \mathbb{S}$ est **vrai** si et seulement s'il y a une chaîne à explorer qui mène de l'état global $\mathcal{S}$ vers l'état global $\mathcal{S}'$, où la dernière synchronisation est étiquetée par l et la condition de blocage atteinte par cette synchronisation vaut μ. Formellement :

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

– Le temps ne peut pas s'écouler dans un état global si une communication urgente est prête, c'est-à-dire une chaîne qui finit soit avec une synchronisation sur une porte urgente ou soit avec une action de communication de la forme « $G\ O_1 \ldots O_n$ **must in** W » où l'échéance de W est atteint. Le prédicat $\textit{relaxed}(S)$, définit sur $\mathbb{S}$, est **vrai** si et seulement si le temps peut s'écouler dans S. Formellement :

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

Les transitions discrètes sont définies par la règle ($\textit{rdv}$) comme suit :

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

où la fonction *label* change une étiquette non ε de $\mathbb{L}_1$ en une étiquette discrète de $\mathbb{L}_2$:

$$
\textit{label}(G\ v_1 \ldots v_n) \stackrel{\text{def}}{=} \begin{cases} G\ v_1 \ldots v_n & \text{si } \textit{tag}(G) = \textbf{visible} \\ \tau & \text{sinon} \end{cases}
$$

Les transitions de temps sont définies par la règle ($\textit{time}$), qui permet l'écoulement de t unités de temps si aucune communication urgente n'est prête. Le nouvel état est calculé par l'augmentation de toutes les phases par t, en utilisant « + », définie par $(\forall u)\ (\theta + t)(u) \stackrel{\text{def}}{=} (t_u + t, \mu_u)$ où $\theta(u) = (t_u, \mu_u)$.

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

Nous illustrons la sémantique par la dérivation de deux transitions de STT de l'exemple sur l'interrupteur de la figure D.2. Nous montrons que si *User* est dans l'état *Rdy* et *Lamp* dans l'état *Low*, alors trois unités de temps peuvent s'écouler avant que l'interrupteur soit utilisé. Formellement : $(\pi, \theta, \varnothing) \xrightarrow{3} (\pi, \theta + 3, \varnothing) \xrightarrow{Push} (\pi \oslash [Lamp \mapsto Bright], \theta, \varnothing)$ où $\pi \stackrel{\text{def}}{=} [User \mapsto Rdy, Lamp \mapsto Low]$, et $\theta \stackrel{\text{def}}{=} [User \mapsto (0, \mathbf{f}), Lamp \mapsto (0, \mathbf{f})]$ (ou « $\mathbf{f}$ » est bref pour « **faux** »).

D'abord, $(\pi, \theta, \varnothing) \xrightarrow{3} (\pi, \theta + 3, \varnothing)$ est dérivé comme suit :

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

Après, $(\pi, \theta + 3, \varnothing) \xrightarrow{Push} (\pi \oslash [Lamp \mapsto Bright], \theta, \varnothing)$ est dérivé comme suit :

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

La prémisse $(act(Rdy), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)$ est dérivée comme suit, où il est rappelé que $act(Rdy) = (\textbf{wait}\ 1;\ Push\ \textbf{;}\ \textbf{to}\ Rdy)$:

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

Finalement, la prémisse $(Push;\ \textbf{to}\ Rdy, (2, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Rdy, (2, \mathbf{f}), \varnothing)$ est dérivée comme suit :

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

La prémisse $(act(Low), (3, \mathbf{f}), \varnothing) \stackrel{Push}{\Longrightarrow} (Bright, (3, \mathbf{f}), \varnothing)$ est dérivé de manière semblable avec les règles $(comm)$, (to), (seq_1) et $(select)$.

Avec notre approche sémantique, nous respectons la propriété standard que le temps doit s'écouler avec la même vitesse dans toutes les unités. En plus, la proposition suivante montre que la sémantique respecte les propriétés souhaitées qui sont présentés dans la définition D.2.

Proposition D.1. Le STT qui correspond à la sémantique d'une spécification ATLANTIF satisfait les propriétés de (i) additivité de temps, (ii) déterminisme de temps et (iii) progrès maximal des actions urgentes.

D.5 Représenter des constructions de haut-niveau

Dans cette section, deux exemples des constructions haut-niveau sont présentés, suivi par des traductions (manuelles) en ATLANTIF. Ceci est une illustration de la puissance d'expression d'ATLANTIF.

D.5.1 Latence

Dans le langage RT-LOTOS [48], la construction de *latence* représente une extension des limiteurs de vie, qui ne peut pas être exprimé dans la majorité d'autres langages de haut-niveau.

Le fragment du code RT-LOTOS suivant exprime un cas typique de latence :

```
...
hide G in
  (delay(3) latency(2) G ;
  stop)
...
```

Sémantiquement, cet exemple commence par l'écoulement de trois unités de temps (instruction « **delay**(3) »). Puis, la communication par G peut avoir lieu à n'importe quel instant pendant les deux unités de temps suivantes (« **latency**(2) G »). Comme G est caché (« **hide** »), cette communication devient une action τ. Remarquons que l'urgence qui est normalement impliquée par le **hide** n'influence pas le temps de latence ; au lieu de cela, G ne devient urgent qu'après l'écoulement des deux unités de temps.

Traduction vers ATLANTIF

Déclarer G comme synchroniseur **urgent** n'est pas praticable, donc G est déclaré caché (**hidden**). Pour forcer la communication au dernier moment possible, après une période où elle est optionnelle, nous utilisons la modalité **must** dans l'action de communication, ce que donne le fragment de code ATLANTIF suivant :

```
...
sync G  :hidden is U1 end sync
...
unit U1 is
  from S1
    wait 3 ;
    G must in [0,2] ;
    stop
end unit
...
```

Dans la Section 5.4.2, nous montrons informellement que cette traduction est correcte.

D.5.2 Gestion des exceptions

Quelques langages de haut-niveau sont définis avec des constructions pour exprimer la gestion des exceptions, c'est-à-dire interruption d'un comportement, qui est alors suivi par un autre comportement, appelé le traitement de l'exception.

Dans LOTOS NT, une exception est levée par l'instruction **raise** et le traitement de l'exception est décrit par une instruction **trap**. Un exemple simple est donné par le fragment du code LOTOS NT de Fig. 5.12 sur page 121, où x est une variable entière, G une porte et B un comportement arbitraire.

Ce code commence par vérifier si la variable x est entre deux et dix, et dans ce cas, x est réduite par un. Sinon, l'exception *EX1* est levée, paramétrée par x.

Le traitement de l'exception *EX1* est juste une communication par G, qui émit la valeur du paramètre. Le comportement régulier aussi que le traitement de l'exception sont suivis par B.

Traduction vers ATLANTIF

Dans Fig. 5.14 sur page 123 nous montrons des fragments d'une module ATLANTIF qui correspondent au code LOTOS NT de Fig. 5.12. Le comportement normal est représenté par l'unité *Main_Unit*, le traitement de l'exception est représenté par l'unité *EH* et B est représenté par l'unité *Unit_B*. Nous supposons que les unités *Main_Unit* et *EH* sont toujours démarrées au même instant, et que la variable x est définie dans une unité supérieure à *Main_Unit*. Quand *Main_Unit* devient active (et *EH* avec), elle exécute la même vérification sur x comme le fragment LOTOS NT : si x est entre deux et dix, alors elle est réduite par un, suivi par une synchronisation de *Main_Unit* et *EH* sur *Normal_Termination*. Ce synchroniseur est une construction auxiliaire qui représente la fin de la portée de l'exception. La levée d'une exception est représentée par le synchroniseur silencieux *EX1*.

Dans l'unité *EH*, l'état discret *S1* représente un état *veille* qui attend qu'une exception a lieu ou sinon pour être arrêté par *Normal_Termination*. Quand une exception a lieu, *Main_Unit* est arrêtée et *EH* passe à l'état discret *S2*, qui représente le traitement de l'exception original, y inclus la communication par G. La synchronisation sur *Termination_After_Exception* représente la fin du traitement de l'exception.

Dans la Section 5.7.2, nous montrons informellement que cette traduction est correcte.

D.6 Traduire ATLANTIF en modèles graphiques

Nous avons développé un outil prototype qui traduit des modules ATLANTIF vers les automates temporisés (TA) de l'outil UPPAAL [89] et vers les réseaux de Petri temporisés

(TPN) de l'outil TINA [16]. Cette section décrit les idées centrales de ces traductions. Nous supposons que le lecteur connaît les notations des TA UPPAAL et des TPN TINA.

Restrictions communs

Certains concepts d'ATLANTIF ne peuvent être traduits ni vers les TA UPPAAL, ni vers les TPN TINA. Concrètement, les modèles ATLANTIF doivent utiliser du temps dense, les expressions dans les actions **wait** et dans les fenêtres de temps doivent être des constantes entiers, les affectations non déterministes ne sont pas admis, les motifs doivent être composés des variables ou des constants. En plus, la traduction vers les TA ne prend pas encore en compte des boucles **while**, bien qu'une telle traduction serait faisable. L'élimination des synchroniseurs silencieux n'existe pas dans UPPAAL et dans TINA ; nous les traduisons alors par des transitions non étiquetées, ce que change la sémantique.

Traduction vers UPPAAL

Chaque unité ATLANTIF est traduit par un TA. Chaque état discret s est traduit par une location du TA (appelé s aussi) et un invariant est généré à partir des contraintes **must** de la transition à branchement multiple dont l'origine est s. L'action $act(s)$ est décomposée en une transition du TA pour chacun de ces chemins. Si la communication par une porte permet plusieurs ensembles de synchronisation qui contiennent l'unité actuelle, alors le chemin est scindé pour donner une transition pour chaque ensemble de synchronisation. Puisque les TA ne permettent pas des offres de communication, les échanges de données sont émulés par des variables partagées.

Un défi clé est que la synchronisation des TA d'UPPAAL est limité à deux automates, pendant qu'ATLANTIF permet des synchronisations « multi-way » entre $n > 2$ unités. Pour notre solution, il est nécessaire qu'exactement une unité émet des données (toutes les offres sont des émissions), pendant que les $(n-1)$ autres unités reçoivent des données (toutes les offres sont des réceptions). Alors la communication dans l'émetteur est scindé en une séquence de $(n-1)$ communications, dont chacune synchronise avec un récepteur.

En plus, il faut émuler de démarrer et d'arrêter les unités. Pour cet objectif, chaque TA qui correspond à une unité qui est arrêtée ou démarrée par au moins un synchroniseur est munie d'une transition location nommée « *disabled* ». La séquence des communications dans une « unité d'émission » décrite en-dessus est prolongée par une communication étiquetée par un canal de diffusion (« broadcast ») « G_stop ! » si le synchroniseur arrête des unités, et par une communication étiquetée par un canal de diffusion « G_start ! » si le synchroniseur démarre des unités. Chaque unité qui est arrêtée par G reçoit une transition supplémentaire vers la location « *disabled* », étiqueté « G_stop ? », à partir de chaque location. Chaque unité qui est démarrée par G reçoit une transition supplémentaire étiqueté « G_start ? » entre « *disabled* » et la location initiale.

Traduction vers TINA

Pour chaque unité d'ATLANTIF un TPN est crée. Tout état discret s est traduit par une place de TPN (appelé s aussi) et l'action $act(s)$ est décomposée en une transition du TA, étiquetée par une porte. En ce qui concerne les contraintes temporelles, nous ne considérons que les intervalles et nous implémentons une solution inspirée par [20], qui a besoin des places et des transitions supplémentaires. Étant donné une communication par une porte G qui correspond à une transition T dans le TPN, nous calculons la somme m de tous les délais des actions **wait** avant la communication. Nous supprimons ces actions et nous augmentons les bornes de l'intervalle par m. L'intervalle obtenue est alors implémenté sous forme de zéro, une ou deux transitions comme suit :

– Si la borne inférieur de l'intervalle est $n > 0$, alors nous ajoutons une transition non étiquetée avec la contrainte temporelle « $[n, \omega[$ » (ou « $]n, \omega[$ », si la borne est stricte), aucune place de sortie, et une nouvelle place d'entrée s_1. Nous ajoutons s_1 aux places inhibiteurs de T et aux places de sortie de toute transition dont s est déjà une place de sortie.
– Si la modalité de la communication est **may** et si la borne supérieure de l'intervalle est n, alors nous ajoutons une transition non étiquetée avec la contrainte temporelle « $]n, \omega[$ » (ou « $[n, \omega[$ », si la borne est stricte), aucune place de sortie, et une nouvelle place d'entrée s_2. Nous ajoutons s_2 aux places d'entrée de T et aux places de sortie de toute transition dont s est déjà une place de sortie. En plus, la nouvelle transition est prioritaire par rapport à T.
– Si la modalité de la communication est **must** et si la borne supérieure de l'intervalle est n, alors nous ajoutons une transition non étiquetée avec la contrainte temporelle « $[n, n]$ », aucune place de sortie, et une nouvelle place d'entrée s_3. Nous ajoutons s_3 aux places d'entrée de T et aux places de sortie de toute transition dont s est déjà une place de sortie. En plus, la nouvelle transition est moins prioritaire par rapport à T et les autres transitions auxiliaires pour T.

Le TPN qui correspondent au module à traduire est composé des TPN qui correspondent à chacune des unités. Si une transition dans le TPN composé est étiquettée par une porte dont le synchroniseur arrête des unités, alors les places de sortie de cette unité sont supprimées de la transition. Si le synchroniseur démarre des unités, alors les places initiales de ces unités sont ajoutées aux places de sortie de cette transition.

Implémentation

Notre avons implémenté (par la méthode proposé dans [62]) nos traducteurs dans un outil nommé `atlantif` (au présent 2 193 lignes de code SYNTAX, 13 146 lignes de code LOTOS NT et 538 lignes de code C). L'architecture d'`atlantif` est illustrée dans Fig. D.3, où les flèches grasses représentent les traducteurs proposés par notre outil.

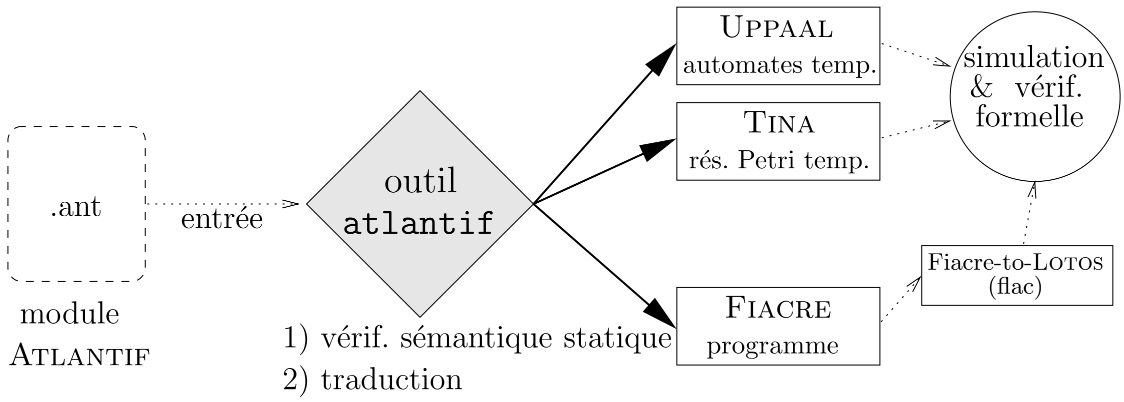

Fig. D.3 – Schéma pour l'utilisation de l'outil prototype `atlantif`

D.7 Exemple : Un ascenseur

Fig. 7.1 sur le page 182 contient le code Atlantif d'un ascenseur inspiré d'un exemple semblable dans [78]. Cet exemple couvre à la fois données, concurrence et temps-réel.

Nous avons utilisé notre outil `atlantif` pour traduire cette module vers un TPN de Tina, qui est décrit dans Fig. 7.2 (code) et dans Fig. 7.3 (description graphique).

L'utilisation de l'outil `selt` de la boîte d'outils Tina permet de découvrir deux fautes dans le code de Fig. 7.1, et de les corriger ainsi.

D.8 Conclusion

D.8.1 Contribution

Caractéristiques du langage

Dans la section D.1, nous avons développé l'objectif de définir un modèle intermédiaire avec une grande puissance d'expression par rapport à la gestion des données, la concurrence, et le temps réel, ce qu'est atteint par Atlantif comme suit :

– La *gestion des données* d'Atlantif est fortement basée sur le modèle Ntif. Nous n'avons pas eu besoin de définir des extensions sur cet aspect, car Ntif est déjà capable de représenter la gestion des données de haut-niveau.
– Par rapport à la *concurrence*, Atlantif inclut la notion des synchroniseurs, qui permet d'une manière concise, puissante et intuitive la représentation des synchronisations possibles entre unités dans une composition parallèle.
  Des constructions puissantes de haut-niveau telles que la gestion des exceptions sont représentables en Atlantif (avec la préservation de la sémantique), grâce à la notion des synchroniseurs silencieux (« **silent** »).
– La syntaxe *temps réel* d'Atlantif introduit (sur le niveau des unités) les constructions de haut-niveau d'une action de délai indépendante et d'une extension limiteur de vie

à l'action de communication ; ATLANTIF introduit en plus (au-dessus du niveau des unités) la possibilité de déclarer une porte comme **urgente**. Sur les deux niveaux, des constructions avec des limites fortes aussi qu'avec des limites faibles sont définies.
Dans la définition formelle de la sémantique, nous avons introduit la nouvelle notion de phase, comme un élément clé assurer facilement la satisfaction des propriétés sémantiques généralement souhaitées.

Nous avons définis une sémantique formelle dans la section D.4 par des règles détaillées, ce que souligne les conséquences de la représentation combinée des données, de temps réel et de la concurrence dans un seul modèle : par exemple, la distinction de plusieurs cas dans le calcul d'une phase après une synchronisation (prédicat $next_\theta$ sur le page 276) est nécessaire à cause de la combinaison entre le temps réel et la concurrence. En plus, il peut être observé que des petits simplifications dans la syntaxe peuvent considérablement simplifier la sémantique (voir la version simplifiée d'ATLANTIF dans [118].

Comparaison systématique d'ATLANTIF avec d'autres modèles

D'autres modèles intermédiaires ont été définis, souvent avec des objectifs similaires à ceux d'ATLANTIF. Dans cette section, nous faisons une comparaison systématique entre ATLANTIF et des approches similaires.

Le résumé de cette comparaison se trouve dans la table D.2, qui considère des approches centrales de la syntaxe et de la sémantique.

Pour complémenter la table D.2, nous discutons dans la suite les différences principales entre ATLANTIF et FIACRE, BIP et MODEST.

- FIACRE et ATLANTIF appliquent tous les deux une structuration des processus par des transitions à branchement multiple, une idée héritée de NTIF. Donc les deux modèles sont très semblables sur le niveau de processus (respectivement unité), sauf par rapport à la syntaxe temporelle, qui existe dans les unités d'ATLANTIF, mais non pas dans les processus de FIACRE.
  Au-dessus du niveau de processus, FIACRE définit des priorités parmi les portes de synchronisation, ce que n'existe pas dans ATLANTIF. En plus, FIACRE permet de définir des intervalles de temps associés aux portes de synchronisation, pendant que ATLANTIF permet sur ce niveau seulement la distinction « **visible/hidden** » et entre « **urgent/silent** ».
  Par ailleurs, il y a une grande différence entre FIACRE et ATLANTIF par rapport à la représentation de la synchronisation : FIACRE utilise des vecteurs de synchronisation qui combinent plusieurs identificateurs de processus et plusieurs identificateurs de portes dans une seule construction ; et dans ATLANTIF, les portes sont définies séparement par l'utilisation d'un synchroniseur pour chacune. Notre traduction de section D.6 montre que l'approche d'ATLANTIF n'est pas plus expressive, mais à notre avis, cette traduction indique aussi que nos formules de synchronisation sont une notation plus concise et plus intuitive.
- BIP permet des variables hybrides et des priorités ; ces deux concepts n'existent pas dans ATLANTIF. Les notations pour les synchronisations dans les deux modèles sont

	domaine de temps		limites								synchronisation				
	dense	discrète	faibles	fortes	horloges	limiteurs de vie	construction délai	variables globales	variables hybrides	trans. ht-niv.	simple	complexe	start/stop	priorités	probabilités
IF-2.0	+	−	+	+	−	+	−	+	−	−	−	−	+	(+)	−
ALTARICA	+	−	+	+	+	+	−	−	−	−	+	+	−	+	−
NTIF	−	−						−	−	+	−	−	−	−	−
FIACRE	+	−	−	+	−	+	−	+	−	+	+	+	−	+	−
BIP	−	+	+	+	+	+	−	−	+	−	+	+	−	+	−
MODEST	+	−	+	+	+	+	−	+	−	+	+	−	−	−	+
ATLANTIF	+	+	+	+	−	+	+	−	−	+	+	+	+	−	−

TAB. D.2 – Comparison des modèles intermédiaires

très différentes, et aussi d'une puissance d'expression différente : démarrer et arrêter des processus (unités) n'existe qu'en ATLANTIF, pendant que seulement BIP contient une construction pour une communication de diffusion (« *broadcast* ») ; d'où aucune des deux notations n'est plus expressive que l'autre.
Sur le niveau des processus, BIP applique des transitions « *condition/action* » avec donc une syntaxe qui est plus simple mais moins concise, et elle est surtout plus loin des langages de haut-niveau que les transitions à branchement multiple d'ATLANTIF.
– MODEST élargit le choix non déterministe avec des probabilités ; nous avons choisi de ne pas introduire une telle construction dans ATLANTIF. Une autre différence au niveau des processus est que ATLANTIF structure ses unités par des états discrets, pendant que les processus de MODEST utilisent des structures de contrôle semblables aux langages de haut-niveau, où des états discrets ne sont pas nécessaires. Par contraste, le temps dans des processus MODEST est implémenté par des variables d'horloges comme dans les automates temporisés, pendant qu'Atlantif applique des constructions de haut-niveau (« **wait** » et limiteur de vie avec les communications).
L'idée de plusieurs ensembles de synchronisation qui est utilisé dans ATLANTIF n'existe pas dans l'opérateur de composition parallèle de MODEST, ni la possibilité de démarrer et d'arrêter des processus.

Nous observons donc que dans la conception des différents modèles intermédiaires, des réponses très différentes ont été trouvées à la question « Quelle combinaison de constructions des langages de haut-niveau et des modèles graphiques est optimale pour le dépouillement, la puissance expressive et l'intuition ? ». En particulier, aucun entre ces modèles peut facilement être identifier d'être plus proche aux langages (ou aux modèles) que les autres.

ATLANTIF a fait au moins une décision d'une manière différente que chacun des autres modèles[36], et donc ce modèle propose une combinaison unique des caractéristiques.

L'extension des possibilités pour appliquer la vérification formelle

Par les traductions que nous avons décrites dans la section D.6 il est possible d'appliquer la vérification formelle sur des sous-ensembles d'ATLANTIF qui sont suffisamment grands. Pour ces traductions, nous avons définis des émulations pour les constructions qui n'ont pas de correspondance directe dans le modèle cible, en particulier les suivantes :

– Pour UPPAAL, nous avons émulé la synchronisation « multiway », les offres et de démarrer et d'arrêter des unités.
– Pour TINA, nous avons émulé les limites faibles et les contraintes temporelles sur le niveau des processus.

En particulier, nous considérons la représentation du temps réel d'ATLANTIF dans le concept de temps réel bien différent des réseaux de Petri temporelles (dans le dialecte de TINA) d'être une approche intéressante.

[36] À remarquer que cet énoncé reste correct par rapport aux sous-ensembles d'ATLANTIF qui peuvent être traduits vers des outils de vérification.

Les sous-ensembles d'ATLANTIF qu'on sait traduire couvrent des parties différentes de notre langage. En pratique, cela nous permet dans beaucoup de cas (ce que nous avons pu observé dans les sections D.6 et D.7) de trouver une traduction appropriée pour un système donné. Donc le « sous-ensemble vérifiable » d'ATLANTIF est considérablement grand.

L'intention centrale de cette thèse a été de rapprocher les deux tâches de la modélisation expressive et concise et de la vérification formelle, par une extension de la classe des systèmes qui peuvent être vérifiés en pratique. Nous considérons avoir atteint cet objectif, pour les raisons suivantes :

- Par rapport aux langages de haut-niveau (comme ceux discutés en section D.3.3), ATLANTIF fait une extension des possibilités de vérification, grâce à sa combinaison des éléments du syntaxe de haut-niveau avec un lien vers les outils de vérification. Un tel lien est rarement défini pour des langages de haut-niveau, au moins pour ceux qui combinent la concurrence avec le temps réel et la gestion des données.
- Par rapport aux autres modèles intermédiaires, ATLANTIF fait une extension des possibilités de vérification, parce que, comme nous l'avons observé au début de la section D.8, notre modèle les complète en proposant une combinaison des caractéristiques syntaxiques nouvelle et unique. Bien que, comme nous l'avons vu, le sous-ensemble traduisible d'ATLANTIF ait une puissance d'expression ni plus grande, ni plus petite que les autres modèles, ses constructions sont conçues pour être très proches aux algèbres de processus modernes, tels que E-LOTOS et LOTOS NT ; donc ATLANTIF représente un complément convenable.
- Par rapport aux modèles graphiques, ATLANTIF fait une extension des possibilités de vérification, grâce à ses constructions concises (telles que synchroniseurs, offres et transitions à branchement multiple) qui permettent aux utilisateurs une description facile des systèmes qui seraient difficile à décrire directement dans un modèle graphique. Par l'application d'une de nos traductions, la vérification formelle devient ainsi possible.
  Bien sûr, la puissance d'expression des sous-ensembles d'ATLANTIF qu'on sait traduire vers des modèles graphiques ne peut pas être supérieur à la puissance d'expression de ces modèles. Mais un utilisateur d'UPPAAL par exemple qui souhaite modéliser un système avec de la synchronisation « multiway » ou un utilisateur de TINA qui souhaite modéliser un système avec des limites faibles ne pourraient pas facilement trouver une représentation correcte dans ces modèles. Par contre, avec ATLANTIF, l'utilisateur peut facilement implémenter ces aspects et faire une traduction automatique vers le modèle souhaité.

D.8.2 Perspective

Sémantique dynamique – constructions séquentielles. Ils existent plusieurs axes, sur lesquels le développement d'ATLANTIF pourrait être poursuit :

- Des nouvelles constructions syntaxiques peuvent être introduites, telles que des *priorités* (par exemple, entre des différentes portes) ou la synchronisation de diffusion (par exemple, comme celle d'UPPAAL).

- La syntaxe actuelle d'Atlantif pourrait être modifiée. En particulier, il est possible de remplacer la structure stricte des unités avec une structure qui permet d'*instancier dynamiquement* les unités, à partir d'une unité générique paramétrable.
- La sémantique actuelle d'Atlantif pourrait être modifiée. En particulier, les restrictions de la sémantique statique qui ont des origines plutôt techniques (par exemple que les actions wait ne sont pas permis après les communications) pourraient être levées dans une version révisée.

Sémantique dynamique – constructions séquentielles. Il y a des différentes possibilités pour le futur développement des traductions :

- Certaines restrictions qui définissent les sous-ensembles d'Atlantif pour lesquels des traducteurs sont définis pourraient être levées dans une version révisée ; par exemple concernant les boucles **while**.
- Les traducteurs pourraient être élargis pour cibler d'autres dialectes des automates temporisés et des réseaux de Petri temporisés que ceux d'Uppaal et de Tina respectivement, ou même d'autres modèles graphiques. De tels nouveaux traducteurs sont sensés s'ils augmentent le sous-ensemble d'Atlantif qui peut être traduit.

Résumé

La validation des systèmes critiques réalistes nécessite d'être capable de modéliser et de vérifier formellement des données complexes, du parallélisme asynchrone, et du temps-réel simultanément.

Des langages de haut-niveau, comme ceux qui héritent des fondations théoriques des algèbres de processus, ont une syntaxe concise et une grande expressivité pour représenter ces aspects. Cependant, ils disposent de peu d'outils logiciels permettant d'appliquer des algorithmes efficaces du model-checking. Néanmoins, de tels outils existent pour des modèles graphiques, de niveau plus bas, tels que les automates temporisés (par exemple Uppaal) et les réseaux de Petri temporisés (par exemple Tina).

Les modèles intermédiaires sont un moyen pour combler le fossé qui sépare les langages des modèles graphiques. Par exemple, Ntif (*New Technology Intermediate Format*) a été proposé pour représenter des processus séquentiels non temporisés qui manipulent des données complexes. Dans cette thèse, nous proposons un nouveau modèle nommé Atlantif, qui enrichit Ntif de constructions temps-réel et de compositions parallèles de processus séquentiels. Leur synchronisation est exprimée d'une manière simple et intuitive par la nouvelle notion de *synchroniseur*.

Nous montrons qu'Atlantif est capable d'exprimer les constructions principales des langages de haut-niveau. Nous présentons aussi des traducteurs d'Atlantif vers des automates temporisés (pour la vérification avec Uppaal) et vers des réseaux de Petri temporisés (pour la vérification avec Tina). Ainsi, Atlantif étend la classe des systèmes qui peuvent en pratique être vérifiés formellement, ce que nous illustrons par un exemple.

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

Abstract

The validation of real-life critical systems raises the challenge of being able to formally model and verify complex data, asynchronous concurrency, and real-time aspects simultaneously.

High-level languages, such as those inheriting from the theoretical foundations of process algebras, provide a concise syntax and a high expressive power regarding these aspects. Yet, they lack software tools enabling the application of efficient model checking algorithms. On the other hand, such tools exist for graphical, lower level, models such as timed automata (e.g., Uppaal) and time Petri nets (e.g., Tina).

Intermediate models are a key to bridge the gap between languages and graphical models. For instance, Ntif (*New Technology Intermediate Format*) was proposed to represent untimed sequential processes that handle complex data. In this thesis, we propose a new model named Atlantif, which extends Ntif with real-time constructs and parallel compositions of sequential processes. Their synchronization is expressed in a simple and intuitive way using the new notion of *synchronizers*.

We show that Atlantif is capable of expressing the main constructs of high-level languages. We also present translators from Atlantif to timed automata (for verification using Uppaal) and to time Petri nets (for verification using Tina). Thus, Atlantif extends the class of systems that can practically be formally verified, which we illustrate along an example.

Keywords: automaton, concurrency, formal method, intermediate model, Petri net, process algebra, real time, verification