

*Institut de Recherche en Communication et Cybernétique de Nantes  
Université de Nantes*

## **On the Formal Verification of EFFBD Models Using a Structural Translation to Time Petri Nets**

**Charlotte Seidner      Olivier H. Roux**

Institut de Recherche en Communication et Cybernétique de Nantes  
Université de Nantes  
1 rue de la Noë, BP 92 101  
44321 Nantes CEDEX 3, France

Phone: +33 2 40 37 69 22, Fax: +33 2 40 37 69 30  
E-Mail: {seidner,o-h.roux}@ircsyn.ec-nantes.fr

*Technical Report RI2007-3 3695*

October 2007  
Revised April 2008



# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Informal presentation of EFFBDs</b>	<b>3</b>
1.1 Introduction . . . . .	3
1.2 Parallel structures and replications . . . . .	3
1.3 Selection structures . . . . .	4
1.4 Loop and iteration structures . . . . .	5
1.5 Functions . . . . .	5
1.6 Data and resources modeling with items . . . . .	6
1.7 Sub-scenarios and decomposed functions . . . . .	7
<b>2 Formal description of EFFBDs</b>	<b>9</b>
2.1 Syntax of EFFBDs . . . . .	10
2.1.1 Syntax of untimed EFFBDs . . . . .	10
2.1.2 Syntax of timed EFFBDs . . . . .	11
2.1.3 Syntax of EFFBDs with forced terminations . . . . .	11
2.1.4 Syntax of EFFBDs with AA mode . . . . .	12
2.1.5 Implicit nodes and structures . . . . .	12
2.2 Additional definitions . . . . .	12
2.2.1 Predecessors, successors and paths . . . . .	12
2.2.2 Decomposed function and exit branches . . . . .	13
2.2.3 Parallel and selection structures . . . . .	13
2.2.4 Loop and iteration structures . . . . .	14
2.2.5 Well-formed EFFBDs . . . . .	15
2.2.6 Item production and consumption . . . . .	15
2.3 Behavioral semantics of EFFBDs . . . . .	16
2.3.1 Semantics of untimed EFFBDs . . . . .	16
2.3.2 Semantics of timed EFFBDs . . . . .	18
2.3.3 Semantics of EFFBDs with forced terminations . . . . .	19
2.3.4 Semantics of EFFBDs with AA mode . . . . .	21
<b>3 Translation of an EFFBD to a TPN</b>	<b>24</b>
3.1 Syntax and semantics of Petri nets . . . . .	24
3.1.1 Petri nets . . . . .	24
3.1.2 Time Petri nets . . . . .	25
3.1.3 Time Petri nets with reset arcs . . . . .	26

3.2	Translation principles . . . . .	27
3.2.1	Item pattern . . . . .	27
3.2.2	Control patterns . . . . .	27
3.3	Translation patterns . . . . .	27
3.3.1	Parallel structures . . . . .	28
3.3.2	Selection structures . . . . .	28
3.3.3	Iteration structures . . . . .	28
3.3.4	Loop structures . . . . .	29
3.3.5	Non decomposed functions . . . . .	29
3.3.6	Sub-scenarios . . . . .	29
3.3.7	Node pattern summation . . . . .	30
3.4	Construction of the complete net . . . . .	31
3.4.1	Places and transitions . . . . .	31
3.4.2	Additional arcs . . . . .	32
<b>4</b>	<b>Properties and results</b>	<b>34</b>
4.1	Non-reentrance and boundedness of the EFFBDs . . . . .	34
4.2	Strong timed bisimulation . . . . .	35
4.3	Additional results . . . . .	37
	<b>Conclusion</b>	<b>39</b>

# List of Figures

1	Model checking of high-level behavioral models . . . . .	1
1.1	Example of an EFFBD ( <i>adapted from [Lon95]</i> ) . . . . .	4
1.2	Example of a replication structure and its parallel structure equivalent . . . . .	4
1.3	Nested loops . . . . .	5
1.4	A consuming/producing function . . . . .	7
1.5	A decomposed function and its sub-scenario . . . . .	8
1.6	An equivalent single-level model . . . . .	8
2.1	Non-decomposed, multi-exit function and its single-exit equivalent . . . . .	10
2.2	Implementation of data broadcasting . . . . .	11
3.1	A simple Petri net . . . . .	24
3.2	A simple time Petri net with a reset arc . . . . .	27
3.3	Patterns of nodes $i \in AND_{in}$ and $o \in AND_{out}$ . . . . .	28
3.4	Patterns of nodes $i \in AND_{in}$ and $o \in AND_{out}$ with a kill branch . . . . .	28
3.5	Patterns of nodes $i \in OR_{in}$ and $o \in OR_{out}$ . . . . .	28
3.6	Patterns of nodes $i \in IT_{in}$ and $o \in IT_{out}$ . . . . .	29
3.7	Patterns of nodes $i \in LP_{in}$ , $o \in LP_{out}$ and $le \in LE$ . . . . .	29
3.8	Patterns of function $F$ (non-AA mode) . . . . .	29
3.9	Patterns of function $F$ (AA mode) . . . . .	30
3.10	Patterns of decomposed multi-exit function $FD$ . . . . .	30
4.1	Example of a bounded EFFBD (model of a bounded buffer) . . . . .	35

## Abstract

This technical report describes the EFFBD (Enhanced Function Flow Block Diagram) formalism, widely used to model and analyze the behavior of complex engineering systems. The report formally establishes the syntax and behavioral semantics of EFFBDs. It also proposes a structural translation of EFFBDs to time Petri nets (TPNs); this translation is then proved as preserving the behavioral semantics through timed bisimilarity.

After proving results on the boundedness of the resulting TPNs, it became possible to extend a number of fundamental properties (such as liveness, state-access etc. decidability) from bounded TPNs to the so-called “bounded EFFBDs”.

*Keywords:* Systems Engineering, time Petri nets, embedded system design and modeling, formal verification, timed bisimulation.

## Résumé

Ce rapport technique décrit le formalisme des EFFBDs (*Enhanced Function Flow Block Diagrams*), largement employé pour modéliser et analyser le comportement de systèmes d'ingénierie complexes. La syntaxe et la sémantique comportementale des EFFBDs y sont décrites de manière formelle. En outre, ce document propose une traduction structurelle des EFFBDs vers les réseaux de Petri temporels (*time Petri nets*, TPNs) dont on prouve qu'elle préserve la sémantique comportementale au sens de la bisimulation temporelle.

Ayant prouvé des résultats sur le caractère borné des réseaux obtenus, il a été possible d'étendre un certain nombre de propriétés fondamentales (telles que la décidabilité de la vivacité, de l'accessibilité d'états etc.) établies sur les TPNs bornés vers les «EFFBDs bornés».

*Mots-clés :* Ingénierie des Systèmes, réseaux de Petri temporels, modélisation et conception de systèmes embarqués, vérifications formelles, bisimulation temporelle.

# Introduction

## Usefulness and usability of formal methods in Systems Engineering processes

Systems Engineering (SE) is defined by the INCOSE<sup>1</sup> as an “interdisciplinary approach” to perform the “realization of successful systems”, from the definition of “customer needs and required functionality early in the development cycle” to the “design synthesis and validation” steps [INC04a,INC04b]. Application fields are quite numerous and include defense, aerospace engineering, road or railroad transport, computer science etc.

Amongst the tasks assigned to the systems engineer lies *verification* (“Does the systems meets some given specifications?”) and *validation* (“Does the system accomplish its intended requirements?”) so as to ensure such properties as safety and dependability [IEE94]. The development of ever larger and more complex systems has made the assessment of such properties most essential. However, the resort to usual, simple methods (such as performing simulations on a model of the system) is then insufficient as exhaustive testing would be very long, if not impossible.

On the other hand, *model checking* may address the shortcomings of the simulation method. The core concept behind model checking is the following question: “given a formal expression of a property,  $\varphi$ , and a formal model of the system,  $S$ , does  $S$  verify  $\varphi$ ?”, which is also denoted  $S \models \varphi$ . Model checking is an automated process, based on the model state-space exploration, and whose algorithms heavily rely on the mathematical properties of the model classes describing  $S$  and  $\varphi$ .

Moreover, when considering the high-level behavioral models commonly used in SE, it appears more efficient first to supply them with a formal behavioral semantics, then to translate them into formal, lower-level models on which model checking techniques and results are well established, rather than to develop specific methods, along with a possibly complex body of theory. The process is illustrated Fig. 1.

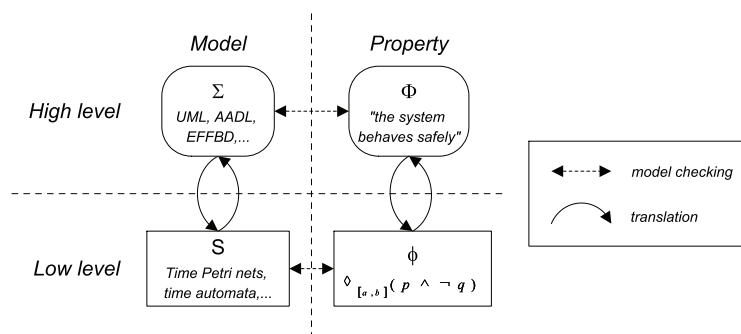


Figure 1: Model checking of high-level behavioral models

Since the translations are obviously designed so as to preserve the system behavior with regards to the class of properties to check, this procedure indeed benefits from the usually most powerful tools and results obtained on the lower-level models to assess high-level properties.

The work presented in this report focuses on *Enhanced Function Flow Block Diagrams* or EFFBDs [US 68], a graphical formalism widely used in SE projects developed by major companies such as AIRBUS or

<sup>1</sup> International Council on Systems Engineering.

the NASA [Lon95]. Although no formal semantics was established (to our knowledge), this model has proved to be consistent and mature over the last decades and its implementation in various design and modeling tools provides a consistent semantics *de facto*. As for the lower-level model, time Petri nets or TPNs [Mer74] appeared well-suited to the EFFBD structure.

## Related works

Over the last decades, a large number of research efforts have been focused on safety and dependability methods and techniques. In the software engineering field, in particular, it led to the formalization of such modeling languages as UML (see for instance [BMIS04]).

In SE (whose application field is even broader than software engineering), similar projects have been emerging since the last few years. Amongst the modeling tools used or developed within the SE community, one should for instance cite the *Systems Modeling Language* (SysML, [Sys07]), result of a joint initiative of the Object Management Group (OMG) and the INCOSE<sup>1</sup>. SysML is also widely based on a subset of UML. However, the well-known issue of the lack of precise semantics in UML is still not solved in SysML [VD05].

Recently, the *Architecture Analysis and Design Language* (AADL, [SAE04]) has gained importance as a powerful modeling tool by allowing a number of formal analyses [Rug05]. However, it appears that the language still suffers from semantical imprecisions. For instance, whereas some dynamical aspects of AADL are described with hybrid automata, others are not covered, thus allowing different interpretations. Moreover, AADL still lacks the maturity of such models as EFFBDs which go back, in their simplest form, to the 1950's.

Lastly, the author of [Her04] proposed a semi-formal semantics for FFBDs (a subset of EFFBDs) and a translation to Petri nets (PNs) preserving the “causal chain representation” i.e. a non temporal bisimilarity. However, no formal proof is given for the translation; in addition, FFBDs are strictly less expressive than EFFBDs as they cannot represent data flows nor resource usage.

## Our contribution

This report formally establishes the complete syntax and behavioral semantics of EFFBDs. To our knowledge, this formalization was never proposed so far. It also proposes a structural translation of EFFBDs to TPNs; this translation is proved to preserve the behavioral semantics (i.e. timed bisimilarity). After proving results on the boundedness of the resulting TPNs, it was possible to extend a number of fundamental properties (such as the decidability of liveness, state-access etc.) from bounded TPNs to the so-called bounded EFFBDs. These results led to the implementation and integration of an operational formal verification tool within a development platform, used in systems design for defense applications; this tool is not discussed here but is (partially) presented in [SLR07].

## Organization of the report

Chapter 1 is an informal presentation of EFFBDs; the formal semantics is given in Section 2. Chapter 3 presents the translation patterns giving for each EFFBD element its TPN equivalent. Chapter 4 then gives some properties and results obtained on the translation, amongst which the time bisimilarity between models. We conclude this report by presenting our further research work.

---

<sup>1</sup> The specification is actually inspired by the EFFBD formalism.

# Chapter 1

## Informal presentation of EFFBDs

This chapter provides an informal description of the EFFBD formalism, including the most common variations. This description was compiled from [Lon95] and [Vit00]; the material presented here is indeed largely inspired by the implementation realized in the CORE<sup>®</sup> software, a system design platform developed by VITECH CORP.<sup>1</sup>

### 1.1 Introduction

In order to provide an efficient specification of both functional and data control, systems engineers often use relatively simple graphical representations such as EFFBDs. These representation provide the designer with an easy and intuitive framework to describe the behavior of complex, distributed, hierarchical, concurrent and communicating systems. EFFBDs derive from FFBDs, first developed in the late 1950s at TRW CORP. They add to the description of the control flow the data control layer, thus enriching the initial model expressiveness.

More precisely, EFFBDs describe the functions performed by the system and the order in which they are to be executed. This order is specified through the functions dynamic parameters (i.e. their execution duration), control environment (control constructs) and data environment (including resources).

EFFBDs offer a large variety of control constructs such as parallel branches, loops, selection branches etc. The following sections provide an informal presentation of available control structures and data control constructs.

Fig. 1.1 shows an example of an EFFBD; the diagram does not correspond to an actual system but shows the main features of the EFFBD formalism. Note that constructs are usually consisting of two corresponding circular nodes. As a general rule, only “well-nested” diagrams are valid: constructs must be exited in the reverse order they were entered.

In this section, the terms “active” and “enabled” are synonyms<sup>2</sup>, as well as “exited” and “completely executed”.

### 1.2 Parallel structures and replications

A *parallel structure* consists of two AND nodes and  $n$  parallel branches ( $n \geq 2$ ). After the structure has been enabled and entered, the first construct of every branch is enabled. The structure is exited as soon as the last construct on every branch has been completely executed; this rule may induce some synchronization states.

Moreover, any number of parallel branches can hold the *kill* modifier. As soon as the last construct on a kill branch has been exited, the whole AND construct is exited and any active constructs in the parallel

---

<sup>1</sup> <http://www.vitechcorp.com>

<sup>2</sup> Except for functions, for which “active” can either means “enabled” or “executing”, see Chapter 2.



The *selection process*, i.e. the set of rules determining the branch choice, takes various forms in the different implementations such as *selection probabilities* or *internal scripts*, all of which are not part of the native EFFBD formalism. Here, we consider that any branch can be taken, regardless of the system state.

## 1.4 Loop and iteration structures

A *loop construct* consists of two LP nodes surrounding a loop branch. Entering the loop enables the first construct of the loop branch; when the last construct of the branch is exited, the control returns to the loop opening. The behavior thus defined is *infinite*.

In addition, when encountering a LE (*Loop Exit*) node, the control passes to the structure following the closing LP node of the loop construct containing the LE node. In addition, any active constructs contained in the loop structure is terminated. In case of nested loops, the control goes to the closest enclosing loop, as illustrated Figure 1.3 (the annotation under each LE node indicates the reference loop construct). For obvious reasons, LE nodes should not (and therefore *cannot*) be directly contained in iteration or loop structures; moreover, any LE node must be in the same hierarchical level (or *scenario*) as its reference loop construct (see Section 1.7).

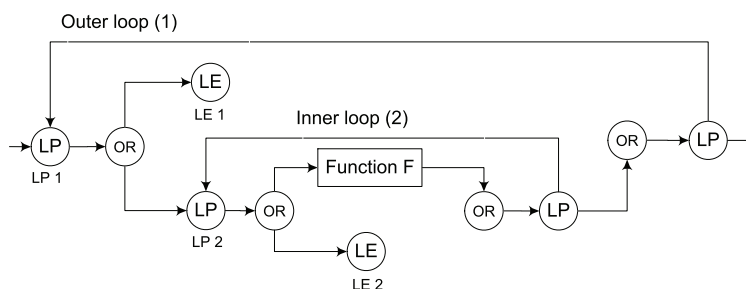


Figure 1.3: Nested loops

An *iteration structure* comprises two IT nodes surrounding an iterated branch, an internal counter and a maximal iteration value  $i_M$  ( $i_M \geq 2$ ). Entering the iteration enables the first construct of the iterated branch and initializes the counter to 1. When the last construct of the branch is exited, two behaviors are possible:

- if the counter value is strictly less than  $i_M$ , the first construct of the iterated branch is enabled again and the counter incremented;
- else, the structure is exited.

## 1.5 Functions

*Function constructs*, represented by rectangular nodes, are the system functions containers, the functions being in turn the model core. Function constructs can be *single-exit* or *multi-exit*, in which case they have  $n$  exit branches ( $n \geq 2$ ) converging on a closing OR node. For multi-exit functions, the choice between the different exit branches is either internal (and corresponds to the selection process described earlier) or described in a sub-scenario (see below). Moreover, the formalism distinguishes *non-decomposed* and *decomposed* functions, the latter being described in Section 1.7.

A function can start its execution if and only if it is both *enabled* (by the control flow) and, for non-decomposed functions, *triggered* (by the item flow). Execution durations are described by probability laws; however, for the sake of simplicity, we consider here that execution durations belong to a time interval  $[\alpha, \beta]$ . When the execution has been completed, the (potential) output items are produced and the function construct is exited.

## 1.6 Data and resources modeling with items

Items either represent *data stores* (e.g. outputs of a sensor), *triggers* (e.g. orders issued by a controller) or *resources* (e.g. energy, CPU memory, waste etc.). They are linked to functions by arrows on which, for resources, are shown the quantities exchanged. Items are *produced*, *consumed* or *captured* by non-decomposed functions (the latter case simply being a combination of a consumption followed by a production in the same quantity). The correspondence between item nature and influence on the system behavior is detailed hereunder and illustrated on a basic example at the end of this section.

A number of item production and consumption policies can indeed be defined and modeled. For instance, if the considered item models a reading produced by a sensor, it should probably represent only the latest value of the measured quantity. Therefore, a newly produced item *replaces* the older one. On the other hand, a newly produced item that represents a resource should be *added* to the existing ones. In the following, we will adopt the policy implemented in CORE: producing any type of item adds it to the existing quantity. However, the “produce and replace” policy can also be easily implemented and translated.

In addition, in order to model such different situations as sending a message to a number of stations or accessing a common resource, some items (namely data stores and triggers) are *broadcast* across the system whereas resources are *shared* between the receiving functions. In other words, for each non-resource item, there is as much copies as they are functions receiving it.

Concerning item consumption, the wait for an item can be described as *blocking* or *non-blocking*: in the first case only, the function cannot begin its execution as long as the item is not available. In addition, some consuming policies are either *destructive* (i.e. the item is actually consumed) or *non-destructive* (i.e. it is read but not removed from the system).

Finally, only resources can be consumed in (integer) quantities greater than 1: data stores and triggers do not indeed represent the actual value of data but rather the fact that they are produced and available. Therefore, only one data store or trigger can be exchanged at a time. An additional rule, known as *Acquire Available* (or AA) and implemented in CORE, is defined for resources: in this policy, the function enabled by the control flow acquires the amount of resources currently available and waits for the remainder of the resource to become available. In the non-AA mode, the function waits for the full amount of needed resource to become available before consuming it.

Table 1.1 summarizes the behavior of items according to their nature. Note that, due to the broadcasting rule, an item can be both data store and trigger (the nature changes from the receiving function point of view) but neither both data store and resource nor both trigger and resource. Moreover, as it has no influence on the system behavior, data store consumption shall not be further considered.

	<b>Data Store</b>	<b>Trigger</b>	<b>Resource</b>
<b>Broadcast</b>	yes	yes	no
<b>Blocking wait</b>	no	yes	yes
<b>Destructive consumption</b>	no	yes	yes
<b>Quantity</b>	1	1	at least 1
<b>Acquire Available</b>	no	no	possible

Table 1.1: Item producing and consuming policies

Item consumption is *synchronous*, except when dealing with resources in the AA mode. The behavior of a receiving/producing function is defined by the following algorithm:

1. wait for the control flow;
2. once enabled, consume, in the available quantities, the input resources marked as AA, until all needed quantities are consumed;
3. once all remaining resources (non-AA) and triggers are available, consume them and begin function execution;
4. end function execution and produce the output items.

As an illustration, let us consider a function  $F$ , shown Fig. 1.4, that becomes enabled by the control flow at  $t_0$ ; it requires 3 units of resource  $rA$  (in the AA mode), 2 units of resource  $rB$  (in the non-AA mode) and is triggered by  $tC$ . Its execution is 5 time units (t.u.) long and it produces 4 units of resource  $rC$ . Initially (i.e. at  $t_0$ ), 2 units of  $rA$  are available and at  $t_0 + 3$  t.u., another 2 units become available.  $rB$  is initially unavailable; 3 units are available from  $t_0 + 1.25$  t.u. Finally,  $tC$ , initially lacking, is produced at  $t_0 + 2.5$  t.u.

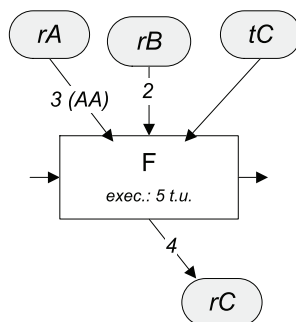


Figure 1.4: A consuming/producing function

The time-line describing the function behavior is given in Table 1.2.

	Function events	System events
$t_0$	enable function consume 2 units of $rA$	2 units of $rA$ available
$t_0 + 1.25$		3 units of $rB$ available
$t_0 + 2.5$		$tC$ produced
$t_0 + 3$	consume 1 unit of $rA$ consume 2 units of $rB$ and $tC$ , begin execution	2 units of $rA$ available
$t_0 + 8$	end execution, produce 4 units of $rC$	

Table 1.2: Example of a function execution time-line

Finally, when two (or more) functions need a common resource at the same moment, in respective quantities resulting in a conflict, no priority is defined to resolve it: a random choice is performed to decide which function gets the resource. In our future work, we intend to implement some priority management such as FIFO queues<sup>1</sup>.

## 1.7 Sub-scenarios and decomposed functions

As stated before, EFFBDs support a hierarchical description of the system, by using *sub-scenarios* linked to decomposed functions. These sub-scenarios can be brought down to a simplifying and compact way of designing the model.

Decomposed functions can also be single- or multi-exited; in the latter case, the use of EXIT nodes ending each branch of the sub-scenario indicates which exit branch of the upper level is the target (the single-exit case is trivial and is not considered hereunder). Moreover, when encountering an EXIT node, any active constructs in the sub-scenario is terminated. Fig. 1.5 illustrates a decomposed function  $FD$  and its attached sub-scenario. Note the open rectangles in the sub-scenario: they indicate the connection points with the upper level. As an illustration, Fig. 1.6 gives an equivalent single-level model.

Table 1.3 provides a possible step-by-step time-line for the execution of the decomposed function (to ease the reading, no temporal information is provided). Here, we consider that the containing function

<sup>1</sup> Note that FIFO queues on resource access are partially implemented in CORE, on resources consumed in AA mode [Vit00].

$FD$  is active as long as any node in the sub-scenario is active. Therefore,  $FD$  does not appear in the equivalent model.

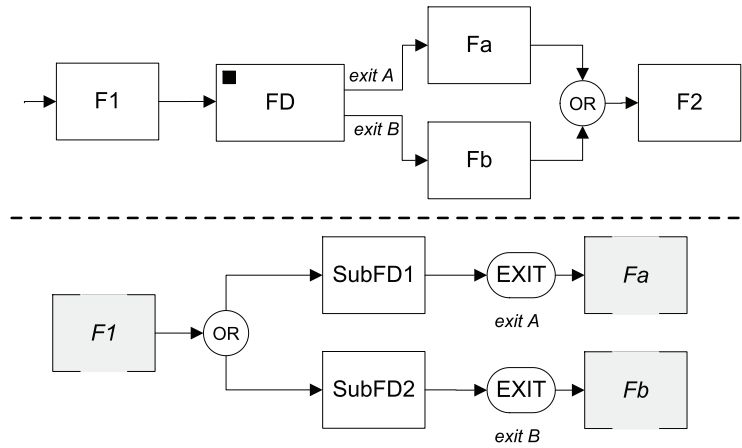


Figure 1.5: A decomposed function and its sub-scenario

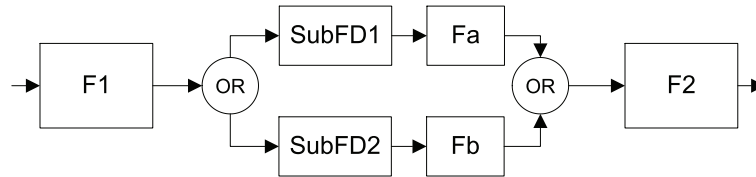


Figure 1.6: An equivalent single-level model

Step	Active nodes
1	F1
2	FD + opening OR
3	FD + SubFD2
4	FD + EXIT to exitB
5	Fb
6	closing OR
7	F2

Table 1.3: Time-line of Fig. 1.5 EFFBD

As for LE nodes, EXIT nodes cannot be directly contained in iteration or loop structures; in addition, their target must be an exit branch of the containing function i.e. in the hierarchical level immediately above. Finally, there must be a 1-to-1 correspondence between EXIT nodes and exit branches in the decomposed functions; this rule is not restrictive but enforces a correct model design.

## Chapter 2

# Formal description of EFFBDs

This chapter presents the syntax and behavioral semantics of EFFBDs, from their untimed, simplest form (i.e. without forced termination features, such as kill modifiers, or AA resource consumption mode) to more complex ones. We thus describe the syntax of untimed EFFBDs and progressively complicate the model by successively taking time, forced termination and AA mode into account (Section 2.1). Some additional definitions are provided in Section 2.2. Finally, Section 2.3 gives the corresponding behavioral semantics; to our knowledge, this semantics has never been formally established.

Most of the formalization presented here and in the following chapters were designed so as to provide a straightforward proof of the equivalence between an EFFBD and its corresponding TPN.

The notations adopted in this report are the following:

- $\mathbb{B} = \{true, false\}$ ;
- $\mathbb{N}$ ,  $\mathbb{Z}$  and  $\mathbb{R}_{\geq 0}$  are respectively the sets of natural integers, integers and positive real numbers,  $\mathbb{N}^*$  (resp.  $\mathbb{Z}^*$ ) is the set  $\mathbb{N} \setminus 0$  (resp.  $\mathbb{Z} \setminus 0$ );
- the usual operators  $+$ ,  $-$ ,  $<$ ,  $\leq$ ,  $\geq$ ,  $>$ ,  $=$  are extended (element-wise) to vectors of  $A^n$  with  $n \in \mathbb{N}$  and  $A = \mathbb{N}, \mathbb{Z}, \mathbb{R}_{\geq 0}$ ;
- given two sets  $A$  and  $B$ ,  $B^A$  is the set of applications from  $A$  to  $B$ ;
- $\vec{0}$  is the null vector;
- $\emptyset$  is the empty set.

Unless otherwise specified, opening brackets stand for “and” conditions and unions of ensembles contain no duplicate elements (i.e.  $\{a\} \cup \{a\} = \{a\}$ ).

The behavioral semantics considered in the following are described as TTS, whose definition is recalled below. TTS are in fact usual transition systems with two types of labels (discrete labels for modeling events and positive reals labels for time elapsing).

**Definition 2.1** (Timed Transition System [HMP92]). *A timed transition system over the set of actions  $\Sigma$  is a tuple  $(Q, Q_0, \Sigma, \rightarrow)$  where  $Q$  is a set of states,  $q_0 \in Q$  the initial state,  $\Sigma$  a finite set of actions disjoint from  $\mathbb{R}_{\geq 0}$  and  $\rightarrow \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$  a set of edges<sup>1</sup>.*

The definition of the strong timed bisimulation, which shall be needed in Section 4.2, is recalled below.

**Definition 2.2** (Strong timed bisimulation [Par81]). *Let  $S_1 = (Q_1, q_0^1, \Sigma, \rightarrow_1)$  and  $S_2 = (Q_2, q_0^2, \Sigma, \rightarrow_2)$  be two TTS and  $\sim$  a binary relation<sup>2</sup> over  $Q_1 \times Q_2$ .  $\sim$  is a strong timed bisimulation between  $S_1$  and  $S_2$  if:*

---

<sup>1</sup>  $(q, \bullet, q') \in \rightarrow$  is also written  $q \xrightarrow{\bullet} q'$ .

<sup>2</sup>  $(q_1, q_2) \in \sim$  is written  $q_1 \sim q_2$ .

$$R_0 \quad q_0^1 \sim q_0^2$$

$$R_d \quad \forall q_1, q_1' \in Q_1, q_2 \in Q_2, d \in \mathbb{R}_{\geq 0} \text{ s.t. } q_1 \xrightarrow{d}_1 q_1' \text{ and } q_1 \sim q_2, \exists q_2' \in Q - 2 \text{ s.t. } q_2 \xrightarrow{d}_2 q_2' \text{ and } q_1' \sim q_2'$$

$$R'_d \quad \forall q_2, q_2', q_1, d \text{ s.t. } q_2 \xrightarrow{d}_2 q_2' \text{ and } q_1 \sim q_2, \exists q_1' \text{ s.t. } q_1 \xrightarrow{d}_1 q_1' \text{ and } q_1' \sim q_2'$$

$$R_\sigma \quad \forall q_1, q_1', q_2, \sigma \in \Sigma \text{ s.t. } q_1 \xrightarrow{\sigma}_1 q_1' \text{ and } q_1 \sim q_2, \exists q_2' \text{ s.t. } q_2 \xrightarrow{\sigma}_2 q_2' \text{ and } q_1' \sim q_2'$$

$$R'_\sigma \quad \forall q_2, q_2', q_1, \sigma \text{ s.t. } q_2 \xrightarrow{\sigma}_2 q_2' \text{ and } q_1 \sim q_2, \exists q_1' \text{ s.t. } q_1 \xrightarrow{\sigma}_1 q_1' \text{ and } q_1' \sim q_2'$$

## 2.1 Syntax of EFFBDs

In the following, trivial cases such as replications and sub-scenarios attached to single-exit functions are not taken into account. In addition, we consider here that non-decomposed multi-exit functions are equivalent to a single-exit function followed by a selection structure, as illustrated Fig. 2.1.



Figure 2.1: Non-decomposed, multi-exit function and its single-exit equivalent

### 2.1.1 Syntax of untimed EFFBDs

**Definition 2.3** (Untimed EFFBD). *An untimed EFFBD is a tuple  $\mathcal{E}_U = (\mathcal{N}, \mathcal{I}, \mathcal{A}, \text{count}, n_0, I_0)$  where:*

- $\mathcal{N}$  is a finite, non-empty set of nodes defined as the reunion of the following subsets:
  - $AND_{in}$  and  $AND_{out}$ : the set of opening and closing AND nodes;
  - $OR_{in}$  and  $OR_{out}$ : the set of opening and closing OR nodes;
  - $IT_{in}$  and  $IT_{out}$ : the set of opening and closing IT nodes;
  - $LP_{in}$  and  $LP_{out}$ : the set of opening and closing LP nodes;
  - $FC$ : the set of function constructs<sup>1</sup>.
- $\mathcal{I}$  is a finite set of items; it is the reunion of  $\mathcal{D}$ , the set of data-stores/triggers and  $\mathcal{R}$ , the set of resources ( $\mathcal{D} \cap \mathcal{R} = \emptyset$ );
- $\mathcal{A}$  is a finite, non-empty set of control and item arcs:  $\mathcal{A} = \mathcal{A}_C \cup \mathcal{A}'_C \cup \mathcal{A}_I$  where:
  - $\mathcal{A}_C \subseteq \mathcal{N} \times \mathcal{N}$  is the set of forward control arcs<sup>2</sup>;
  - $\mathcal{A}'_C \subseteq \{IT_{out} \times IT_{in}\} \cup \{LP_{out} \times LP_{in}\}$  is the set of backward control arcs ( $\mathcal{A}_C \cap \mathcal{A}'_C = \emptyset$ );
  - $\mathcal{A}_I \subseteq FC \times \mathcal{I} \times \mathbb{Z}^*$  is the set of item arcs;
- $\text{count} : IT_{in} \rightarrow \mathbb{N} \setminus \{0, 1\}$  is the counter function giving for each iteration construct the maximum number of iterations;
- $n_0 \in \mathcal{N}$  is the initial node;
- $I_0 \in \mathbb{N}^{\mathcal{I}}$  gives the initial item amounts<sup>3</sup>.

<sup>1</sup> Here, none of the functions are decomposed.

<sup>2</sup> Note that  $\forall n \in \mathcal{N}, (n, n) \notin \mathcal{A}_C$ .

<sup>3</sup> Although not recommended for a sound modeling, data-stores and triggers initial amount can be set to strictly positive values.

To make the modeling of data broadcast easier, we consider that when a function  $F$  produces a message  $A$  received by  $n$  functions  $f_i$ , it actually produces  $n$   $A_i$  independent messages, each one being consumed by the corresponding  $f_i$  function, thus “unfolding” the broadcast. Item arcs and initial values are also set accordingly; Fig. 2.2 illustrates this rule.

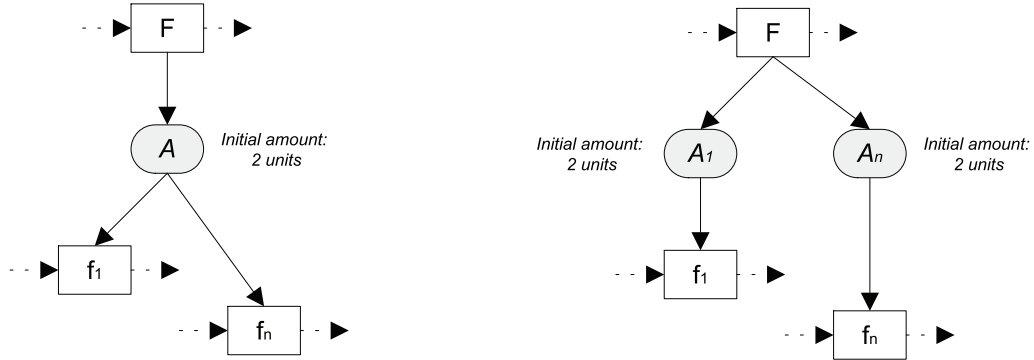


Figure 2.2: Implementation of data broadcasting

Let  $(f, item, k)$  be an element of  $\mathcal{A}_I$ ; if  $k < 0$ ,  $item$  is consumed by  $f$  in the quantity  $k$  and if  $k > 0$ ,  $item$  is produced by  $f$  in the quantity  $k$ ; moreover, if  $item \in \mathcal{D}^1$ ,  $|k| = 1$ .

### 2.1.2 Syntax of timed EFFBDs

**Definition 2.4** (Timed EFFBD). A timed EFFBD is a tuple  $\mathcal{E}_T = (\mathcal{E}_U, \alpha, \beta)$  where:

- $\mathcal{E}_U$  is an untimed EFFBD;
- $\alpha \in \mathbb{N}^{FC}$  and  $\beta \in (\mathbb{N} \cup \{\infty\})^{FC}$  are respectively the lower bound and the upper bound of the function execution duration mappings.

Here, duration bounds are taken in  $\mathbb{N}$ , yet this choice is not restrictive.

### 2.1.3 Syntax of EFFBDs with forced terminations

**Definition 2.5** (EFFBD with forced terminations). An EFFBD with forced terminations is a tuple  $\mathcal{E}_{FT} = (\mathcal{N}', \mathcal{I}, \mathcal{A}, count, n_0, I_0, \alpha, \beta, kill)$  where:

- $\mathcal{N}'$  is a finite, non-empty set of nodes defined as the reunion of the following subsets:
  - $AND_{in}, AND_{out}, OR_{in}, OR_{out}, IT_{in}, IT_{out}, LP_{in}, LP_{out}$  and  $FC$ , defined as above;
  - $LE$ : the set of LE nodes;
  - $DFC$ : the set of decomposed function constructs;
  - $EXIT$ : the set of EXIT nodes.
- $\mathcal{I}, \mathcal{A}, count, n_0, I_0, \alpha$  and  $\beta$  defined as above;
- $kill \in \mathbb{B}^{\mathcal{A}^c}$  is the kill function, giving for each parallel branch its kill status.

A parallel branch starting from an opening AND node but ending on an LE or EXIT node (instead of a closing AND node) cannot hold the kill modifier (as it would be redundant). Therefore, the parallel branch can be reduced to the arc connecting the last construct of the branch to the closing AND node (hence the expression of  $kill$ ).

<sup>1</sup> No arc links a data-store to a function.

### 2.1.4 Syntax of EFFBDs with AA mode

**Definition 2.6** (EFFBD with AA mode). *An EFFBD with AA mode is a tuple  $\mathcal{E}_{AA} = (\mathcal{E}_{FT}, AA)$  where:*

- $\mathcal{E}_{FT}$  is an EFFBD with forced terminations;
- $AA \in \mathbb{B}^{FC \times \mathcal{R}}$  gives for each function and each resource the AA status of the consumption.

Let  $res$  be a resource and  $f$  a function; if  $f$  does not consume  $res$ , then  $AA(f, res) = false$ .

### 2.1.5 Implicit nodes and structures

Usually, closing AND and OR nodes without any predecessors (i.e. where all parallel or selection branches are ended by a LE or an EXIT node) are not graphically represented but are nonetheless part of the model. This convention ensures a consistent definition of control structures as a set of two corresponding nodes.

As for the exit branches of a decomposed function, we consider that any node placed between an EXIT node and the closing OR node is contained in a so-called EXIT/OR construct. Due to the termination property of EXIT nodes, only one exit branch can be activated, despite the fact that EXIT/OR constructs have more than one entry point.

Finally, a fundamental rule to obtain well-formed EFFBDs consists in assessing that each node (excluding, of course, FC nodes) is constitutive of exactly one control construct and that any two constructs are either in sequence or completely nested one in another. Therefore, when considering a decomposed function as illustrated Fig. 1.5, the content of the internal OR structure (formed by the opening OR node and an implicit closing OR node) is SubFD1 and SubFD2 and the content of the EXIT/OR construct is Fa and Fb. The EXIT nodes are contained by the function FD but not by the internal OR structure.

Predecessors, successors and construct contents are formally described in the following section.

## 2.2 Additional definitions

Some additional definitions are provided below in order to obtain simpler semantic expression.

### 2.2.1 Predecessors, successors and paths

**Definition 2.7** (Predecessors and successors of a node). *Let  $\mathcal{N}$  be a set of nodes and  $\mathcal{A}_C$  a set of control arcs. The predecessors of a node  $n \in \mathcal{N}$  form the set  $Pre(n)$  defined by:*

$$Pre(n) = \{n' \in \mathcal{N} \mid (n', n) \in \mathcal{A}_C\}$$

*The successors of a node  $n \in \mathcal{N}$  form the set  $Post(n)$  defined by:*

$$Post(n) = \{n' \in \mathcal{N} \mid (n, n') \in \mathcal{A}_C\}$$

Due to the definition of  $\mathcal{A}_C$  and  $\mathcal{A}'_C$ , the set  $Post(it_{out})$  (where  $it_{out} \in IT_{out}$ ) does not contain the corresponding  $IT_{in}$  node (the same goes for  $LP_{in}$  and  $LP_{out}$ ).

Let us consider the decomposed function illustrated Fig. 1.5 (p. 8). The definitions above are thus applied:

- $Post(F1) = \{or_{in}\}$  where  $or_{in}$  is the opening OR node;
- $Pre(Fa) = \{exit_A\}$  where  $exit_A$  is the EXIT node pointing to Fa;
- $Post(exit_A) = \{Fa\}$ .

In other words, the actual decomposed function is not taken into account but rather the sub-scenario. These definitions are extended to describe the *content* of constructs, so as to formalize the notion of well-formed EFFBDs.

We also define *paths*:

**Definition 2.8** (Paths). Let  $\mathcal{N}$  be a set of nodes,  $\mathcal{A}_C$  a set of forward control arcs,  $n_{from}$  and  $n_{to}$  two (distinct) elements of  $\mathcal{N}$ . The paths from  $n_{from}$  to  $n_{to}$  are defined as the following set:

$$Paths(n_{from}, n_{to}) = ((n_{from}, n_{to}) \cap \mathcal{A}_C) \cup P$$

where  $P$  is the set of paths with more than two nodes:

$$P = \left\{ (n_{from}, n_1, \dots, n_p, n_{to}) \in \mathcal{N}^{p+2}, p \in \mathbb{N}^* \text{ s.t. } \begin{cases} (n_{from}, n_1) \in \mathcal{A}_C \\ \forall 0 < i < p, (n_i, n_{i+1}) \in \mathcal{A}_C \\ (n_p, n_{to}) \in \mathcal{A}_C \end{cases} \right\}$$

### 2.2.2 Decomposed function and exit branches

The content of a decomposed function  $dfc$  (i.e. the nodes of the sub-scenario) is denoted  $Cont(dfc)$  and the reference decomposed function of an exit node  $exit$  is denoted  $Ref(exit)$ . Obviously,  $exit \in Cont(Ref(exit))$ . Moreover, the set of EXIT/OR constructs is defined as follows:

**Definition 2.9** (EXIT/OR constructs). Given  $exit_1, \dots, exit_m \in EXIT$  and  $or_{out} \in OR_{out}$ ,  $exitOr = \langle exit_1, \dots, exit_m, or_{out} \rangle$  is in the set of EXIT/OR structures EXIT/OR if and only if:

$$\exists! dfc \in DFC \text{ s.t. } \forall 1 \leq i \leq m, exit_i \in Cont(dfc) \quad (2.1)$$

$$\text{and } \forall exit_i, \forall n_{First} \in Post(exit_i), \quad n_{First} \in \{or_{out}\} \cup Pre(or_{out}) \quad (2.2)$$

$$\text{or } \exists! n_{Last} \in Pre(or_{out}) \text{ s.t.}$$

$$(exit_i, n_{First}, \dots, n_{Last}, or_{out}) \in Paths(exit_i, or_{out}) \quad (2.3)$$

$$\text{or } n_{First} \in LE \cup EXIT \quad (2.4)$$

$$\text{or } \exists n_{term} \in LE \cup EXIT \text{ s.t. } Paths(n_{First}, n_{term}) \neq \emptyset \quad (2.5)$$

$$\text{and } \forall n_{Last} \in Pre(or_{out}), \exists! i \text{ s.t. } \quad n_{Last} \in Post(exit_i) \quad (2.6)$$

$$\text{or } \exists! n_{First} \in Post(exit_i) \text{ s.t.}$$

$$(exit_i, n_{First}, \dots, n_{Last}, or_{out}) \in Paths(exit_i, or_{out}) \quad (2.7)$$

For each  $i$ ,  $Post(exit_i)$  is limited to one element and propositions 2.2 to 2.5 are mutually exclusive (and so are propositions 2.6 and 2.7 for each node in  $Pre(or_{out})$ ). If proposition 2.4 is true, it means that the exit branch is “empty”. Finally, propositions 2.6 and 2.7 can seem redundant with propositions 2.2 to 2.5 but they ensure that any branch arriving on a closing OR node comes from an EXIT node: all branches with a “free” end come from an opening node (i.e. are necessarily ended by a LE or an EXIT node).

The content of the EXIT/OR construct is thus defined:

**Definition 2.10** (Content of an EXIT/OR construct). The content of an EXIT/OR construct  $exitOr = \langle exit_1, \dots, exit_m, or_{out} \rangle$ , denoted  $Cont(exitOr)$  is the set of nodes  $n \in \mathcal{N}$  such that there exist a path from some  $exit_i$  node to  $n$  but no path from  $or_{out}$  to  $n$ :

$$\forall n \in \mathcal{N}, n \in Cont(exitOr) \Leftrightarrow \begin{cases} \exists i \text{ s.t. } Paths(exit_i, n) \neq \emptyset \\ Paths(or_{out}, n) = \emptyset \end{cases}$$

### 2.2.3 Parallel and selection structures

We propose a definition for parallel structures; it is largely based on Definition 2.9.

**Definition 2.11** (Parallel structures). *Given  $and_{in} \in AND_{in}$ ,  $and_{out} \in AND_{out}$ ,  $and = \langle and_{in}, and_{out} \rangle$  is in the set of parallel structures  $AND$  if and only if:*

$$\begin{aligned}
& \forall n_{First} \in Post(and_{in}) \quad n_{First} \in Pre(and_{out}) \\
& \text{or } \exists ! n_{Last} \in Pre(and_{out}) \text{ s.t.} \\
& \quad (and_{in}, n_{First}, \dots, n_{Last}, and_{out}) \in Paths(and_{in}, and_{out}) \\
& \text{or } n_{First} \in LE \cup EXIT \\
& \text{or } \exists n_{term} \in LE \cup EXIT \text{ s.t. } Paths(n_{First}, n_{term}) \neq \emptyset \\
& \text{and } \forall n_{Last} \in Pre(and_{out}) \quad n_{Last} \in Post(and_{in}) \\
& \text{or } \exists ! n_{First} \in Post(and_{in}) \text{ s.t.} \\
& \quad (and_{in}, n_{First}, \dots, n_{Last}, and_{out}) \in Paths(and_{in}, and_{out})
\end{aligned}$$

Here, the counterpart of proposition 2.4 should never be true in a “well-designed” model as it would mean that the parallel structure is exited as soon as entered, one of the parallel branch being limited to a LE or EXIT node.

The content of a parallel structure is thus defined:

**Definition 2.12** (Content of a parallel structure). *The content of a parallel structure  $\langle and_{in}, and_{out} \rangle$ , denoted  $Cont(\langle and_{in}, and_{out} \rangle)$  is the set of nodes  $n \in \mathcal{N}$  such that there exist a path from  $and_{in}$  to  $n$  but no path from  $and_{out}$  to  $n$ :*

$$\forall n \in \mathcal{N}, n \in Cont(\langle and_{in}, and_{out} \rangle) \Leftrightarrow \begin{cases} Paths(and_{in}, n) \neq \emptyset \\ Paths(and_{out}, n) = \emptyset \\ (n \in EXIT) \Rightarrow (and_{in} \notin Cont(Ref(n))) \end{cases}$$

As explained in Section 2.1.5, an EXIT node is contained by a parallel structure  $and$  only if the full EXIT/OR structure (i.e. including the exit branches) is also contained by  $and$ : therefore,  $and$  cannot be in the same sub-scenario as the EXIT node. The set of selection structures,  $OR$ , and the content of a selection structure  $Cont(or)$  are similarly defined.

## 2.2.4 Loop and iteration structures

**Definition 2.13** (Loops). *Given  $lp_{in} \in LP_{in}$ ,  $lp_{out} \in LP_{out}$ ,  $lp = \langle lp_{in}, lp_{out} \rangle$  is in the set of loop structures  $LP$  if and only if:*

$$\begin{aligned}
& (lp_{out}, lp_{in}) \in \mathcal{A}'_C \\
& \forall n_{First} \in Post(lp_{in}) \quad n_{First} \in Pre(lp_{out}) \\
& \text{or } \exists ! n_{Last} \in Pre(lp_{out}) \text{ s.t.} \\
& \quad (lp_{in}, n_{First}, \dots, n_{Last}, lp_{out}) \in Paths(lp_{in}, lp_{out}) \\
& \text{or } n_{First} \in LE \cup EXIT \\
& \text{or } \exists n_{term} \in LE \cup EXIT \text{ s.t. } Paths(n_{First}, n_{term}) \neq \emptyset \\
& \text{and } \forall n_{Last} \in Pre(lp_{out}) \quad n_{Last} \in Post(lp_{in}) \\
& \text{or } \exists ! n_{First} \in Post(lp_{in}) \text{ s.t.} \\
& \quad (lp_{in}, n_{First}, \dots, n_{Last}, lp_{out}) \in Paths(lp_{in}, lp_{out})
\end{aligned}$$

$IT$ , the set of iteration structures, is likewise defined. The content of a loop is defined as follows.

**Definition 2.14** (Content of a loop). *Let  $lp = \langle lp_{in}, lp_{out} \rangle$  be a loop. The content of  $lp$ , denoted  $Cont(lp)$  is the set of nodes  $n$  such that there exist a path from  $lp_{in}$  to  $n$  but no path from  $lp_{out}$  to  $n$ :*

$$\forall n \in \mathcal{N}, n \in Cont(\langle lp_{in}, lp_{out} \rangle) \Leftrightarrow \begin{cases} Paths(lp_{in}, n) \neq \emptyset \\ Paths(lp_{out}, n) = \emptyset \\ (n \in EXIT) \Rightarrow (lp_{in} \notin Cont(Ref(n))) \end{cases}$$

The content of an iteration structure is likewise defined. Finally, we give the formal definition of the reference loop of a LE node, as presented in Section 1.4.

**Definition 2.15** (Reference loop of an LE node). *Let  $le$  be a LE node and  $lp$  a loop.  $lp$  is the reference loop of  $le$ , which is denoted  $lp = Ref(le)$ , if and only if:*

$$\begin{cases} le \in Cont(lp) \\ \forall lp' = \langle lp'_{in}, lp'_{out} \rangle \in LP, le \in Cont(lp') \Rightarrow lp'_{in}, lp'_{out} \notin Cont(lp) \end{cases}$$

### 2.2.5 Well-formed EFFBDs

The definition of a well-formed EFFBD is given hereunder.

**Definition 2.16** (Well-formed EFFBD). *An EFFBD is well-formed if and only if it respects all following properties:*

- (Construct uniqueness) each node (except in FC, DFC and LE) is constitutive of exactly one structure:

$$\begin{cases} \forall n \in AND_{in}, \exists! \bar{n} \in AND_{out} \text{ s.t. } \langle n, \bar{n} \rangle \in AND \\ \forall \bar{n} \in AND_{out}, \exists! n \in AND_{in} \text{ s.t. } \langle n, \bar{n} \rangle \in AND \end{cases}$$

(same for OR, IT, LP and EXIT/OR structures)

- (Continuity) except for the initial node (and the implicit  $AND_{out}$  and  $OR_{out}$  nodes), each node has at least one predecessor. Likewise, there is only one “final node” in the model; it has no successors:

$$\begin{aligned} \forall n \in \mathcal{N}, Pre(n) = \emptyset & \Leftrightarrow \begin{cases} n = n_0 \\ \text{or } n \in AND_{out} \cup OR_{out} \wedge Post(n) = \emptyset \end{cases} \\ \exists! n_{End} \in \mathcal{N} \setminus LE \cup EXIT & \text{ s.t. } \begin{cases} Post(n_{End}) = \emptyset \\ Pre(n_{End}) \neq \emptyset \end{cases} \end{aligned}$$

- (Imbrication) any two constructs are either in sequence or completely nested one in another:

$\forall c \in AND \cup OR \cup IT \cup LP \cup EXIT/OR$ :

$$\begin{cases} \forall \langle n, \bar{n} \rangle \in AND \cup OR \cup IT \cup LP, n \in Cont(c) \Leftrightarrow \bar{n} \in Cont(c) \\ \forall \langle e_1, \dots, e_m, e_0 \rangle \in EXIT/OR, \forall 0 \leq i, j \leq m, e_i \in Cont(c) \Leftrightarrow e_j \in Cont(c) \end{cases}$$

- any LE node and its (unique) reference loop are in the same hierarchical level:

$$\forall le \in LE, \exists! \langle lp_{in}, lp_{out} \rangle \in LP \text{ s.t. } \begin{cases} Ref(le) = \langle lp_{in}, lp_{out} \rangle \\ (\exists f \in DFC, le \in Cont(f)) \Leftrightarrow \langle lp_{in}, lp_{out} \rangle \in Cont(f) \end{cases}$$

- any path in a sub scenario ends on a LE or on an EXIT node:

$$\forall f \in DFC, \forall n \in Cont(f) \setminus LE \cup EXIT, \exists n_{term} \in LE \cup EXIT \text{ s.t. } Paths(n, n_{term}) \neq \emptyset$$

### 2.2.6 Item production and consumption

To ease the description of item handling, we add the following definitions.

**Definition 2.17** (Consuming relations). *Let FC be a set of function constructs,  $\mathcal{I}$  a set of items and  $\mathcal{A}_I$  a set of item arcs. The consuming relation  $Cons : FC \rightarrow \mathbb{N}^{\mathcal{I}}$  is defined for all  $f \in FC$  and  $A \in \mathcal{I}$  by:*

$$Cons(f)[A] = \begin{cases} |k| & \text{if } \exists k < 0 \text{ s.t. } (f, A, k) \in \mathcal{A}_I \\ 0 & \text{otherwise} \end{cases}$$

The producing relation  $Prod : FC \rightarrow \mathbb{N}^{\mathcal{I}}$  is similarly defined for  $k > 0$ .

## 2.3 Behavioral semantics of EFFBDs

Our choice of expressing the semantics as a TTS was twofold. On one hand, we felt necessary to express the semantics in a way that would lead to a straightforward proof of the bisimulation. Therefore, we were compelled to resort to a lower-level model that would be expressive and powerful enough for our needs. On the other hand, we felt that expressing the semantics as a time automata (TA) or a TPN would be easier to perform but difficult to exploit. For instance, we recall that TAs and TPNs are not comparable (so are transition-TPNs and place-TPNs) and that any analysis based on TAs instead of TPNs (or the other way round) would need “extra translations” (necessarily based on the writing of the model semantics as a TTS). In other words, it seemed necessary to express the semantics as a TTS (which incidentally gives the semantics of both TAs and TPNs). As a consequence, the semantics is quite complex, due to the rich “expressivity” of EFFBDs (that is, to the large number of control structures).

### 2.3.1 Semantics of untimed EFFBDs

The behavioral semantics of untimed EFFBDs is defined as a *Transition System* (TS). The TS *states* are triplets that represent the *node activity*, the iteration counters and the item levels. The activity  $A(n)$  of a node  $n$  takes its value in the set  $\mathbb{A} = \{inactive, enabled, executing, executed\}$  if  $n \in FC$  and in the set  $\mathbb{A}^* = \{inactive, enabled, executed\}$  otherwise. The activity *executed* denotes the fact that a node preceding an  $AND_{out}$  node has finished its own execution but is waiting for the other branches to complete their execution. As functions must be in the *executing* state before becoming *executed* or *inactive*, they are the only nodes that cannot directly transit from *enabled* to *executed*. In addition, in the case of closing IT and LP nodes, the control always comes back to the corresponding opening node, even if (in the iteration case) the proper number of iterations was reached, hence additional constraint on these nodes.

The value of the counter relative to an iteration is defined as the number of times the first construct on the iterated branch was enabled.

In order to make reading easier, the semantic rules are decomposed in propositions  $\mathcal{P}_x$ , according to the nature of the processed node. For instance,  $\mathcal{P}_{gen}$  describes the generic rule “each successor of the processed node is enabled ; other nodes, counters and item levels are not affected”. In addition,  $\mathcal{P}_F$  describes a function execution beginning, whereas  $\mathcal{P}'_F$  describes the execution ending.

**Definition 2.18** (Semantics of an untimed EFFBD). *The semantics of an untimed EFFBD  $\mathcal{E}_U = (\mathcal{N}, \mathcal{I}, \mathcal{A}, count, n_0, I_0)$  is a tuple  $||\mathcal{E}_U|| = (S, s_0, \mathcal{N}, \rightarrow)$  where:*

- $S \subseteq \mathbb{A}^{\mathcal{N}} \times \mathbb{N}^{IT_{in}} \times \mathbb{N}^{\mathcal{I}}$ ;
- $s_0 = (A_0, \bar{0}, I_0)$  is the initial state, where  $A_0(n_0) = enabled$  and  $A_0(n) = inactive$  for  $n \neq n_0$ ;
- $\rightarrow \subseteq S \times \mathcal{N} \times S$  is the (discrete) transition relation, defined  $\forall n \in \mathcal{N}$  by:

$$(A, C, I) \xrightarrow{n} (A', C', I') \Leftrightarrow \begin{cases} A'(n) = NextAct \\ (A(n) = enabled \wedge PreCondition) \vee (n \in FC \wedge A(n) = executing \wedge \mathcal{P}'_F) \end{cases}$$

with:

$$NextAct = \begin{cases} executed & \text{if } \begin{cases} n \notin IT_{out} \cup LP_{out} \\ Post(n) \cap AND_{out} \neq \emptyset \\ A(n) = enabled \Rightarrow n \notin FC \end{cases} \\ executing & \text{if } (n \in FC \wedge A(n) = enabled) \\ inactive & \text{otherwise} \end{cases}$$

$$PreCondition = \begin{cases} n \in AND_{in} \cup OR_{out} \cup LP_{in} \Rightarrow \mathcal{P}_{gen} \\ n \in AND_{out} \Rightarrow \mathcal{P}_{A_o} \\ n \in OR_{in} \Rightarrow \mathcal{P}_{O_i} \\ (n \in IT_{in} \wedge C(n) < count(n)) \Rightarrow \mathcal{P}_{I_i} \\ (n \in IT_{in} \wedge C(n) = count(n)) \Rightarrow \mathcal{P}'_{I_i} \\ n \in IT_{out} \Rightarrow \mathcal{P}_{I_o} \\ n \in LP_{out} \Rightarrow \mathcal{P}_{L_o} \\ n \in FC \Rightarrow \mathcal{P}_F \end{cases}$$

*NextAct* thus describes the resulting activity of the processed node (most of the time, it becomes inactive). Note the conditions for the *executed* cases, as mentioned above. *PreCondition* describes the conditions to fulfill in order to process the node; it is a set of mutually exclusive propositions. Defining  $\mathcal{N}^n$  as the set  $\mathcal{N} \setminus \{n\}$ , the definition of the  $\mathcal{P}_x$  conditions are:

$$\begin{aligned} \mathcal{P}_{gen} &= \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{enabled if } n' \in Post(n) \\ A(n') \text{ otherwise} \end{cases} \\ C' = C, I' = I \end{cases} \\ \mathcal{P}_{A_o} &= \begin{cases} \forall n' \in \mathcal{N}^n \begin{cases} A(n') = \text{executed} \wedge A'(n') = \text{inactive if } n' \in Pre(n) \\ A'(n') = \text{enabled if } n' \in Post(n) \\ A'(n') = A(n') \text{ otherwise} \end{cases} \\ C' = C, I' = I \end{cases} \\ \mathcal{P}_{O_i} &= \begin{cases} \exists n_{Select} \in Post(n) \begin{cases} A'(n_{Select}) = \text{enabled} \\ \forall n_{NoSelect} \in Post(n) \setminus \{n_{Select}\}, A'(n_{NoSelect}) = \text{inactive} \\ \forall n' \in \mathcal{N}^n \setminus Post(n), A'(n') = A(n') \end{cases} \\ C' = C, I' = I \end{cases} \\ \mathcal{P}_{I_i} &= \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{enabled if } n' \in Post(n) \\ A(n') \text{ otherwise} \end{cases} \\ \forall it \in IT_{in} \begin{cases} C'(it) = C(it) + 1 \text{ if } it = n \\ C'(it) = C(it) \text{ otherwise} \end{cases} \\ I' = I \end{cases} \\ \mathcal{P}'_{I_i} &= \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{enabled if } n' \in Post(i_o) \text{ where } \langle n, i_o \rangle \in IT \\ \text{executed if } (n' = i_o) \wedge (Post(i_o) \cap AND_{out} \neq \emptyset) \\ A(n') \text{ otherwise} \end{cases} \\ \forall it \in IT_{in} \begin{cases} C'(it) = 0 \text{ if } it = n \\ C'(it) = C(it) \text{ otherwise} \end{cases} \\ I' = I \end{cases} \\ \mathcal{P}_{I_o} &= \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{enabled if } \langle n', n \rangle \in IT \\ A(n') \text{ otherwise} \end{cases} \\ C' = C, I' = I \end{cases} \\ \mathcal{P}_{L_o} &= \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{enabled if } \langle n', n \rangle \in LP \\ A(n') \text{ otherwise} \end{cases} \\ C' = C, I' = I \end{cases} \\ \mathcal{P}_F &= \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = A(n') \\ C' = C \\ (I \geq Cons(n)) \wedge (I' = I - Cons(n)) \end{cases} \end{aligned}$$

$$\mathcal{P}'_F = \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{enabled if } n' \in \text{Post}(n) \\ A(n') \text{ otherwise} \end{cases} \\ C' = C \\ I' = I + \text{Prod}(n) \end{cases}$$

### 2.3.2 Semantics of timed EFFBDs

The semantics of a timed EFFBD is defined as a (TTS). A *valuation* is a mapping  $\nu \in (\mathbb{R}_{\geq 0})^{FC}$  such that  $\forall f \in FC$ ,  $\nu(f)$  is the time elapsed since  $f$  started its execution. It should be noted that  $\nu(f)$  is only meaningful if the function  $f$  is in execution. The TTS states are quadruplets representing node activity, iteration counters, item levels and valuations.

**Definition 2.19** (Semantics of a timed EFFBD). *The semantics of a timed EFFBD  $\mathcal{E}_T = (\mathcal{N}, \mathcal{I}, \mathcal{A}, \text{count}, n_0, I_0, \alpha, \beta)$  is a tuple  $\|\mathcal{E}_T\| = (S, s_0, \mathcal{N}, \rightarrow)$  where:*

- $S \subseteq \mathbb{A}^{\mathcal{N}} \times \mathbb{N}^{IT_{in}} \times \mathbb{N}^{\mathcal{I}} \times (\mathbb{R}_{\geq 0})^{FC}$ ;
- $s_0 = (A_0, \bar{0}, I_0, \bar{0})$ ,  $A_0$  being defined as above;
- $\rightarrow \subseteq S \times (\mathcal{N} \cup \mathbb{R}_{\geq 0}) \times S$  is the transition relation including a discrete transition relation and a continuous transition relation.
  - The discrete transition relation is defined  $\forall n \in \mathcal{N}$  by:

$$(A, C, I, \nu) \xrightarrow{n} (A', C', I', \nu') \Leftrightarrow \begin{cases} A'(n) = \text{NextAct as defined above} \\ (A(n) = \text{enabled} \wedge \text{PreCondition}^T) \vee (n \in FC \wedge A(n) = \text{executing} \wedge \mathcal{P}'_F{}^T) \end{cases}$$

with:

$$\text{PreCondition}^T = \begin{cases} n \in \text{AND}_{in} \cup \text{OR}_{out} \cup \text{LP}_{in} \Rightarrow \mathcal{P}_{gen}^T \\ n \in \text{AND}_{out} \Rightarrow \mathcal{P}_{A_o}^T \\ n \in \text{OR}_{in} \Rightarrow \mathcal{P}_{O_i}^T \\ (n \in \text{IT}_{in} \wedge C(n) < \text{count}(n)) \Rightarrow \mathcal{P}_{I_i}^T \\ (n \in \text{IT}_{in} \wedge C(n) = \text{count}(n)) \Rightarrow \mathcal{P}'_{I_i}{}^T \\ n \in \text{IT}_{out} \Rightarrow \mathcal{P}_{I_o}^T \\ n \in \text{LP}_{out} \Rightarrow \mathcal{P}'_{L_o}{}^T \\ n \in FC \Rightarrow \mathcal{P}'_F{}^T \end{cases}$$

- The continuous transition relation is defined  $\forall d \in \mathbb{R}_{\geq 0}$  by:

$$(A, C, I, \nu) \xrightarrow{d} (A, C, I, \nu') \Leftrightarrow \begin{cases} \forall n \notin FC, A(n) \neq \text{enabled} \\ \forall n \in FC, A(n) = \text{enabled} \Rightarrow I < \text{Cons}(n) \\ \forall n \in FC, A(n) = \text{executing} \Rightarrow \nu(n) + d \leq \beta(n) \\ \nu' = \nu + d \end{cases}$$

The definition of the  $\mathcal{P}'_x{}^T$  conditions are:

$$\mathcal{P}'_F{}^T = \begin{cases} \mathcal{P}_F \text{ as defined above} \\ \forall f \in FC \begin{cases} \nu'(f) = 0 \text{ if } f = n \\ \nu'(f) = \nu(f) \text{ otherwise} \end{cases} \end{cases}$$

$$\mathcal{P}'_F{}^T = \begin{cases} \mathcal{P}'_F \text{ as defined above} \\ \alpha(n) \leq \nu(n) \leq \beta(n) \end{cases}$$

The other conditions are the same as in the untimed case, though with the addition of the constraint  $\nu' = \nu$ . It should be noted that the definition of the continuous transition relation imposes that time cannot elapse until the system has reached a “stable state” (i.e. no instantaneous transition can be taken).

### 2.3.3 Semantics of EFFBDs with forced terminations

The semantics of an EFFBD with forced terminations is also defined as a TTS and most of the  $\mathcal{P}_x^T$  are reused. This semantics is fairly similar to what precedes: again, the TTS states are quadruplets representing node activity, iteration counters, item levels and valuations; some additional rules inactivate nodes contained in a terminating structures. Note that, in the kill case, the semantics forces all predecessors of the closing AND node (here denoted  $\alpha_o$ ) of the parallel structure (denoted  $\alpha$ ) to be in the *executed* activity. Some nodes cannot be the last construct of a (kill) parallel branch<sup>1</sup>; therefore, we did not define a kill behavior for all node types. In the following, the boolean  $\kappa(n)$ , defined for every node  $n$ , is set to *true* iff  $\exists n' \in AND_{out}$  s.t.  $arc = (n, n') \in \mathcal{A}_C \wedge kill(arc) = true$ .

**Definition 2.20** (Semantics of an EFFBD with forced terminations). *The semantics of an EFFBD with forced terminations  $\mathcal{E}_{FT} = (\mathcal{N}, \mathcal{I}, \mathcal{A}, count, n_0, I_0, \alpha, \beta, kill)$  is a tuple  $||\mathcal{E}_{FT}|| = (S, s_0, \mathcal{N}, \rightarrow)$  where:*

- $S \subseteq \mathbb{A}^{\mathcal{N}} \times \mathbb{N}^{IT_{in}} \times \mathbb{N}^{\mathcal{I}} \times (\mathbb{R}_{\geq 0})^{FC}$ ;
- $s_0 = (A_0, \bar{0}, I_0, \bar{0})$ ;
- $\rightarrow \subseteq S \times (\mathcal{N} \cup \mathbb{R}_{\geq 0}) \times S$  is the transition relation including a discrete transition relation and a continuous transition relation.

– The discrete transition relation is defined  $\forall n \in \mathcal{N}$  by:

$$(A, C, I, \nu) \xrightarrow{n} (A', C', I', \nu') \Leftrightarrow$$

$$\left\{ \begin{array}{l} A'(n) = NextAct \text{ as defined above} \\ (A(n) = enabled \wedge PreCondition^{term}) \vee \left[ A(n) = executing \wedge \begin{cases} \mathcal{P}_F^{tterm} \text{ if } \kappa(n) \\ \mathcal{P}'_F \text{ otherwise} \end{cases} \right] \end{array} \right.$$

with:

$$PreCondition^{term} = \left\{ \begin{array}{l} n \in AND_{in} \cup LP_{in} \Rightarrow \mathcal{P}_{gen}^T \\ n \in AND_{out} \Rightarrow \begin{cases} \mathcal{P}_{A_o}^{tterm} \text{ if } \kappa(n) \\ \mathcal{P}_{A_o}^T \text{ otherwise} \end{cases} \\ n \in OR_{in} \Rightarrow \mathcal{P}_{O_i}^T \\ n \in OR_{out} \Rightarrow \begin{cases} \mathcal{P}_{O_o}^{tterm} \text{ if } \kappa(n) \\ \mathcal{P}_{gen}^T \text{ otherwise} \end{cases} \\ (n \in IT_{in} \wedge C(n) < count(n)) \Rightarrow \mathcal{P}_{I_i}^T \\ (n \in IT_{in} \wedge C(n) = count(n)) \Rightarrow \begin{cases} \mathcal{P}_{I_i}^{tterm} \text{ if } \kappa(it_o) \text{ with } \langle n, it_o \rangle \in IT \\ \mathcal{P}'_{I_i} \text{ otherwise} \end{cases} \\ n \in IT_{out} \Rightarrow \mathcal{P}_{I_o}^T \\ n \in LP_{out} \Rightarrow \mathcal{P}_{L_o}^T \\ n \in FC \Rightarrow \mathcal{P}_F^T \\ n \in LE \Rightarrow \begin{cases} \mathcal{P}_{LE}^{tterm} \text{ if } \kappa(lp_o) \text{ with } \langle lp_i, lp_o \rangle = Ref(n) \\ \mathcal{P}_{LE}^T \text{ otherwise} \end{cases} \\ n \in EXIT \Rightarrow \mathcal{P}_{EXIT}^T \end{array} \right.$$

– The continuous transition relation is defined  $\forall d \in \mathbb{R}_{\geq 0}$  as above.

<sup>1</sup> Namely, nodes in  $AND_{in}$ ,  $OR_{in}$ ,  $IT_{in}$ ,  $LP_{in}$ ,  $LE$  and  $EXIT$ .

The definition of the new  $\mathcal{P}_x^X$  conditions are:

$$\begin{aligned}
\mathcal{P}_{A_o}^{term} &= \left\{ \begin{array}{l} \forall n' \in \mathcal{N}^n \begin{cases} A(n') = \text{executed if } n' \in Pre(n) \\ A'(n') = \text{executed if } n' \in Pre(\alpha_o) \\ A'(n') = \text{enabled if } n' = \alpha_o \\ A'(n') = \text{inactive if } n' \in Cont(\alpha) \setminus Pre(\alpha_o) \\ A'(n') = A(n') \text{ otherwise} \end{cases} \\ \forall it \in IT_{in} \begin{cases} C'(it) = 0 \text{ if } it \in Cont(\alpha) \\ C'(it) = C(it) \text{ otherwise} \end{cases} \\ I' = I \\ \nu' = \nu \end{array} \right. \\
\mathcal{P}_{O_o}^{term} &= \left\{ \begin{array}{l} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{executed if } n' \in Pre(\alpha_o) \\ \text{enabled if } n' = \alpha_o \\ \text{inactive if } n' \in Cont(\alpha) \setminus Pre(\alpha_o) \\ A(n') \text{ otherwise} \end{cases} \\ \forall it \in IT_{in} \begin{cases} C'(it) = 0 \text{ if } it \in Cont(\alpha) \\ C'(it) = C(it) \text{ otherwise} \end{cases} \\ I' = I \\ \nu' = \nu \end{array} \right. \\
\mathcal{P}_{I_i}^{term} &= \left\{ \begin{array}{l} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{executed if } n' \in Pre(\alpha_o) \\ \text{enabled if } n' = \alpha_o \\ \text{inactive if } n' \in Cont(\alpha) \setminus Pre(\alpha_o) \\ A(n') \text{ otherwise} \end{cases} \\ \forall it \in IT_{in} \begin{cases} C'(it) = 0 \text{ if } it \in Cont(\alpha) \\ C'(it) = C(it) \text{ otherwise} \end{cases} \\ I' = I \\ \nu' = \nu \end{array} \right. \\
\mathcal{P}_{LE}^T &= \left\{ \begin{array}{l} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{executed if } (n' = lp_o \wedge Post(lp_o) \cap AND_{out} \neq \emptyset) \\ \text{enabled if } n' \in Post(lp_o) \wedge n' \neq lp_i \\ \text{inactive if } n' \in Cont((lp_i, lp_o)) \\ A(n') \text{ otherwise} \end{cases} \\ \forall it \in IT_{in} \begin{cases} C'(it) = 0 \text{ if } it \in Cont((lp_i, lp_o)) \\ C'(it) = C(it) \text{ otherwise} \end{cases} \\ I' = I \\ \nu' = \nu \end{array} \right. \\
\mathcal{P}_{LE}^{term} &= \left\{ \begin{array}{l} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{executed if } n' \in Pre(\alpha_o) \\ \text{enabled if } n' = \alpha_o \\ \text{inactive if } n' \in Cont(\alpha) \setminus Pre(\alpha_o) \\ A(n') \text{ otherwise} \end{cases} \\ \forall it \in IT_{in} \begin{cases} C'(it) = 0 \text{ if } it \in Cont((lp_i, lp_o)) \cup Cont(\alpha) \\ C'(it) = C(it) \text{ otherwise} \end{cases} \\ I' = I \\ \nu' = \nu \end{array} \right.
\end{aligned}$$

$$\mathcal{P}_{EXIT}^T = \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{enabled if } n' \in Post(n) \\ \text{inactive if } n' \in Cont(Ref(n)) \\ A(n') \text{ otherwise} \end{cases} \\ \forall it \in IT_{in} \begin{cases} C'(it) = 0 \text{ if } it \in Cont(Ref(n)) \\ C'(it) = C(it) \text{ otherwise} \end{cases} \\ I' = I \\ \nu' = \nu \end{cases}$$

$$\mathcal{P}_F'^{term} = \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = \begin{cases} \text{executed if } n' \in Pre(\alpha_o) \\ \text{enabled if } n' = \alpha_o \\ \text{inactive if } n' \in Cont(\alpha) \setminus Pre(\alpha_o) \\ A(n') \text{ otherwise} \end{cases} \\ \forall it \in IT_{in} \begin{cases} C'(it) = 0 \text{ if } it \in Cont(\alpha) \\ C'(it) = C(it) \text{ otherwise} \end{cases} \\ I' = I + Prod(n) \\ \nu' = \nu \end{cases}$$

### 2.3.4 Semantics of EFFBDs with AA mode

The semantics of an EFFBD with AA mode is also a TTS; however, the states are extended to represent the levels of resources consumed in the AA mode (*localI*). Due to the subsequent TPN modeling, we suppose here that resource consumption is performed *one unit at a time* (of course, resource consumption is still instantaneous). In addition, a function can now *stay* enabled if some of its input items are missing (hence the modification of *NextAct*).

**Definition 2.21** (Semantics of an EFFBD with AA mode). *The semantics of an EFFBD with AA mode  $\mathcal{E}_{AA} = (\mathcal{N}, \mathcal{I}, \mathcal{A}, count, n_0, I_0, \alpha, \beta, kill, AA)$  is a tuple  $\|\mathcal{E}_{AA}\| = (S, s_0, \mathcal{N}, \rightarrow)$  where:*

- $S \subseteq \mathbb{A}^{\mathcal{N}} \times \mathbb{N}^{IT_{in}} \times \mathbb{N}^{\mathcal{I}} \times \mathbb{N}^{FC \times \mathcal{R}} \times (\mathbb{R}_{\geq 0})^{FC}$ ;
- $s_0 = (A_0, \bar{0}, I_0, localI_0, \bar{0})$  with  $localI_0(f, res) = 0$  for each  $f \in FC$  and  $res \in \mathcal{R}$ ;
- $\rightarrow \subseteq S \times (\mathcal{N} \cup \mathbb{R}_{\geq 0}) \times S$  is the transition relation including a discrete transition relation and a continuous transition relation.
  - The discrete transition relation is defined  $\forall n \in \mathcal{N}$  by:

$$(A, C, I, localI, \nu) \xrightarrow{n} (A', C', I', localI', \nu') \Leftrightarrow$$

$$\left[ \begin{array}{l} A'(n) = NextAct^{AA} \\ (A(n) = \text{enabled} \wedge PreCondition^{AA}) \vee \left[ A(n) = \text{executing} \wedge \begin{cases} \mathcal{P}_F'^{AA-term} \text{ if } \kappa(n) \\ \mathcal{P}_F'^{AA-T} \text{ otherwise} \end{cases} \right] \end{array} \right]$$

with:

$$NextAct^{AA} = \begin{cases} \text{enabled if} & \begin{cases} n \in FC \\ \exists res \in \mathcal{R} : AA(n, res) \wedge local(n, res) < Cons(n) [res] \end{cases} \\ \text{executed if} & \begin{cases} n \notin IT_{out} \cup LP_{out} \\ Post(n) \cap AND_{out} \neq \emptyset \\ A(n) = \text{enabled} \Rightarrow n \notin FC \end{cases} \\ \text{executing if} & \begin{cases} n \in FC \\ A(n) = \text{enabled} \\ \forall it \in \mathcal{D}, I(it) \geq Cons(n) [it] \\ \forall res \in \mathcal{R} \begin{cases} localI(n, res) \geq Cons(n) [res] & \text{if } AA(n, res) \\ I(res) \geq Cons(n) [res] & \text{otherwise} \end{cases} \end{cases} \\ \text{inactive} & \text{otherwise} \end{cases}$$

and

$$PreCondition^{AA} = \begin{cases} n \in AND_{in} \cup LP_{in} \Rightarrow \mathcal{P}_{gen}^{AA} \\ n \in AND_{out} \Rightarrow \begin{cases} \mathcal{P}_{A_o}^{AA-term} & \text{if } \kappa(n) \\ \mathcal{P}_{A_o}^{AA-T} & \text{otherwise} \end{cases} \\ n \in OR_{in} \Rightarrow \mathcal{P}_{O_i}^{AA} \\ n \in OR_{out} \Rightarrow \begin{cases} \mathcal{P}_{O_o}^{AA-term} & \text{if } \kappa(n) \\ \mathcal{P}_{gen}^{AA} & \text{otherwise} \end{cases} \\ (n \in IT_{in} \wedge C(n) < count(n)) \Rightarrow \mathcal{P}_{I_i}^{AA} \\ (n \in IT_{in} \wedge C(n) = count(n)) \Rightarrow \begin{cases} \mathcal{P}'_{I_i}^{AA-term} & \text{if } \kappa(it_o) \\ \text{with } \langle n, it_o \rangle \in IT \\ \mathcal{P}'_{I_i}^{AA-T} & \text{otherwise} \end{cases} \\ n \in IT_{out} \Rightarrow \mathcal{P}_{I_o}^{AA} \\ n \in LP_{out} \Rightarrow \mathcal{P}_{L_o}^{AA} \\ n \in FC \wedge (\forall res \in \mathcal{R}, AA(n, res) = \text{false}) \Rightarrow \mathcal{P}_F^{AA} \\ n \in FC \wedge (\exists res \in \mathcal{R}, AA(n, res) = \text{true}) \Rightarrow \mathcal{P}_F^{AA'} \\ n \in LE \Rightarrow \begin{cases} \mathcal{P}_{LE}^{AA-term} & \text{if } \kappa(lp_o) \text{ with } \langle lp_i, lp_o \rangle = Ref(n) \\ \mathcal{P}_{LE}^{AA-T} & \text{otherwise} \end{cases} \\ n \in EXIT \Rightarrow \mathcal{P}_{EXIT}^{AA} \end{cases}$$

– The continuous transition relation is defined  $\forall d \in \mathbb{R}_{\geq 0}$  by:

$$(A, C, I, localI, \nu) \xrightarrow{d} (A, C, I, localI, \nu') \Leftrightarrow \begin{cases} \forall n \notin FC, A(n) \neq \text{enabled} \\ \forall n \in FC, A(n) = \text{enabled} \Rightarrow \begin{cases} \exists it \in \mathcal{I} \text{ s.t. } I(it) < Cons(n) [it] \\ \text{or } \exists res \in \mathcal{R} \text{ s.t. } \begin{cases} AA(n, res) = \text{true} \\ localI(n, res) < Cons(n) [res] \\ I(res) = 0 \end{cases} \end{cases} \\ \forall n \in FC, A(n) = \text{executing} \Rightarrow \nu(n) + d \leq \beta(n) \\ \nu' = \nu + d \end{cases}$$

The definition of the new  $\mathcal{P}_x^X$  conditions are:

$$\mathcal{P}_{A_o}^{AA-term} = \begin{cases} \mathcal{P}_{A_o}^{term} \\ \forall f \in FC, \forall res \in \mathcal{R}, localI'(f, res) = \begin{cases} 0 & \text{if } f \in Cont(\alpha) \\ localI(f, res) & \text{otherwise} \end{cases} \end{cases}$$

$\mathcal{P}_{O_o}^{AA-term}$ ,  $\mathcal{P}_{I_i}^{AA-term}$ ,  $\mathcal{P}_{LE}^{AA-term}$  and  $\mathcal{P}_F^{AA-term}$  are similarly defined. In addition,

$$\mathcal{P}_F^{AA'} = \begin{cases} \forall n' \in \mathcal{N}^n, A'(n') = A(n') \\ C' = C \\ AsynchCons \vee SynchCons \\ \nu' = \nu \end{cases}$$

with :

$$AsynchCons = \begin{cases} \forall it \in \mathcal{D}, I'(it) = I(it) \\ \exists res \in \mathcal{R} \text{ s.t. } AA(n, res) \wedge localI(n, res) < Cons(n)[res] \wedge I(res) > 0, \\ \quad \forall res' \in \mathcal{R} \text{ s.t. } res' \neq res, \\ \quad \begin{cases} I'(res) = I(res) - 1 \\ I'(res') = I(res') \end{cases} \\ \quad \begin{cases} localI'(n, res) = localI(n, res) + 1 \\ localI'(n, res') = localI(n, res') \end{cases} \end{cases}$$

$$SynchCons = \begin{cases} \forall it \in \mathcal{D} \begin{cases} I(it) \geq Cons(n)[it] \\ I'(it) = I(it) - Cons(n)[it] \end{cases} \\ \forall res \in \mathcal{R} \text{ s.t. } AA(n, res) = true, \begin{cases} localI(n, res) = Cons(n)[it] \\ localI'(n, res) = 0 \end{cases} \\ \forall res' \in \mathcal{R} \text{ s.t. } AA(n, res) = false, \begin{cases} I(res') \geq Cons(n)[res'] \\ I'(res') = I(res') - Cons(n)[res'] \\ localI'(n, res) = 0 \end{cases} \end{cases}$$

All other conditions are similar to the precedent case, with the addition of the constraint  $localI' = localI$ .

## Chapter 3

# Translation of an EFFBD to a TPN

After briefly presenting the Petri net formalism, this chapter gives every translation pattern applied for the structural transformation of any EFFBD into a time Petri net. More comprehensive presentations of TPNs can be found in numerous places (see for instance [Mer74, Mag07]).

### 3.1 Syntax and semantics of Petri nets

Petri nets were first described in [Pet62]; the formalism, both graphical and mathematical, is often used for the modeling, design and analysis of dynamic, discrete, distributed and communicating systems with such diverse application fields as communication protocol modeling, real-time software design etc.

In the preceding chapter, EFFBDs were first introduced in their simplest form, the model being gradually made more complex. Likewise, we give here the syntax and semantics of (untimed) Petri nets (3.1.1) before presenting a temporal extension (3.1.2) and *reset* arcs (3.1.3). The semantics described here corresponds to the single-server, intermediate case (see for instance [BCH<sup>+</sup>05] for a discussion on the different TPN semantics).

#### 3.1.1 Petri nets

Informally speaking, a TPN is a set of *places* and *transitions*, connected by *weighted arcs*. Places are usually represented by circles, transitions by rectangles and arcs by arrows (see Fig. 3.1). While places represent *states* of the modeled system (e.g. “the processor is idle” or “the function is executed”), transitions model the *events* that bring the system from a state to another (e.g. “an alarm is received” or “the function execution is completed”).

Places may contain any number of tokens; they may indicate a level of resources or the validation of a condition: for instance, a processor is active when there is at least one token in the corresponding place and idle if this place is empty.

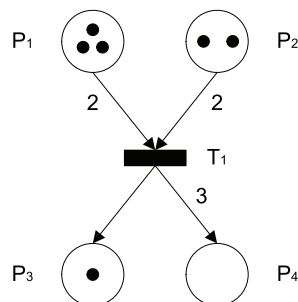


Figure 3.1: A simple Petri net

**Definition 3.1** (Petri net). A Petri net is a tuple  $\mathcal{P} = (P, T, \bullet(\cdot), (\cdot)^\bullet, M_0)$  where:

- $P = \{p_1, \dots, p_m\}$  is a finite, non empty set of places;
- $T = \{t_1, \dots, t_n\}$  is a finite, non empty set of transitions;
- $\bullet(\cdot) \in (\mathbb{N}^P)^T$  is the backward incidence function;
- $(\cdot)^\bullet \in (\mathbb{N}^P)^T$  is the forward incidence function;
- $M_0 \in \mathbb{N}^P$  is the initial marking of the net.

A marking  $M$  of the net is an element of  $\mathbb{N}^P$  such that  $\forall p \in P, M(p)$  is the number of tokens in the place  $p$ . A transition  $t$  is said to be *enabled* by the marking  $M$  if  $M \geq \bullet(t)$ ; it is denoted by  $t \in \text{enabled}(M)$ .

Petri net semantics is defined as a Transition System:

**Definition 3.2** (Semantics of a Petri net). The semantics of a Petri net  $\mathcal{P}$  is defined as a TS  $\|\mathcal{P}\| = (Q, M_0, T, \rightarrow)$  where:

- $Q \subseteq \mathbb{N}^P$ ;
- $\rightarrow \subseteq Q \times T \times Q$  is the transition relation, defined by:

$$\forall t \in T, M \xrightarrow{t} M' \Leftrightarrow \begin{cases} t \in \text{enabled}(M) \\ M' = M - \bullet(t) + (\cdot)^\bullet \end{cases}$$

Let us consider the net illustrated Fig. 3.1; the initial marking is  $M_0 = \{3, 2, 1, 0\}$ . Firing  $T_1$  leads to the marking  $\{1, 0, 2, 3\}$ .

### 3.1.2 Time Petri nets

Labeled *T-time Petri nets* (which will be simply known as *time Petri nets* or TPNs in the following) form a timed extension of the classical formalism in which transitions are fired within a given time interval. The definition provided here refers to integer temporal bound, but this assumption is not restrictive.

**Definition 3.3** (Labeled time Petri nets). A labeled time Petri net is a tuple  $\mathcal{T} = (\mathcal{P}, a, b, A, \Lambda)$  where:

- $\mathcal{P} = (P, T, \bullet(\cdot), (\cdot)^\bullet, M_0)$  is a Petri net;
- $a \in (\mathbb{N})^T$  and  $b \in (\mathbb{N} \cup \{\infty\})^T$  are functions giving for each transition respectively its earliest and latest firing times ( $a \leq b$ );
- $A$  is a set of actions;
- $\Lambda : T \rightarrow A$  is the labeling function.

The preceding notations are still used; in addition, a transition  $t$  is said to be *fireable* when it has been enabled for at least  $\alpha(t)$  time units. A transition  $t_k$  is said to be *newly enabled* by firing transition  $t_i$  from the marking  $M$ , which is denoted by  $\uparrow \text{enabled}(t_k, M, t_i)$ , if the transition is enabled by the new marking  $M - \bullet(t_i) + (\cdot)^\bullet$  but was not by  $M - \bullet(t_i)$ . Formally,

$$\uparrow \text{enabled}(t_k, M, t_i) = (\bullet(t_k) \leq M - \bullet(t_i) + (\cdot)^\bullet) \wedge [(t_k = t_i) \vee (\bullet(t_k) > M - \bullet(t_i))]$$

By extension, the set of transitions newly enabled by firing transition  $t_i$  from the marking  $M$  is denoted  $\uparrow \text{enabled}(M, t_i)$ .

TPNs semantics is defined as a TTS, where states are defined as the association of both a marking  $M$  and a vector of *clock valuations*  $v$ ,  $v(t)$  representing the time elapsed since transition  $t$  was enabled.

**Definition 3.4** (Semantics of a TPN). *The semantics of a labeled time Petri net  $\mathcal{T}$  is defined as a TTS  $\|\mathcal{T}\| = (Q, q_0, \Lambda(T), \rightarrow)$  such that:*

- $Q \subseteq \mathbb{N}^P \times (\mathbb{R}_{\geq 0})^T$ ;
- $q_0 = (M_0, \bar{0})$ ;
- $\rightarrow \subseteq Q \times \{\Lambda(T) \cup \mathbb{R}_{\geq 0}\} \times Q$  is the transition relation including a discrete transition relation and a continuous transition relation.
  - The discrete transition relation is defined by:

$$\forall t \in T \quad (M, v) \xrightarrow{\Lambda(t)} (M', v') \Leftrightarrow \begin{cases} t \in \text{enabled}(M) \\ a(t) \leq v(t) \leq b(t) \\ M' = M - \bullet(t) + (t)\bullet \\ \forall t_k \in T, v'(t_k) = \begin{cases} 0 & \text{if } t_k \in \uparrow \text{enabled}(M, t) \\ v(t_k) & \text{otherwise} \end{cases} \end{cases}$$

- The continuous transition relation is defined by:

$$\forall d \in \mathbb{R}_{\geq 0} \quad (M, v) \xrightarrow{d} (M, v') \Leftrightarrow \begin{cases} v' = v + d \\ \forall t_k \in T, t_k \in \text{enabled}(M) \Rightarrow v'(t_k) \leq b(t_k) \end{cases}$$

### 3.1.3 Time Petri nets with reset arcs

TPNs as defined in Def. 3.3 cannot easily model resets; therefore, reset arcs were added to the classical formalism [DFS98]. These arcs link a place to a transition (graphically, the arrow is replaced by a black diamond); a reset arc does not influence the enabling of the transition but, when it is fired, the place is emptied. More formally:

**Definition 3.5** (Labeled time Petri nets with reset arcs). *A labeled time Petri net with reset arcs is a tuple  $\mathcal{T} = (P, T, \bullet(\cdot), (\cdot)\bullet, \blacklozenge(\cdot), M_0, \alpha, \beta, A, \Lambda)$  where:*

- $P, T, \bullet(\cdot), (\cdot)\bullet, M_0, \alpha, \beta, A, \Lambda$  are defined as above;
- $\blacklozenge(\cdot) \in (\{0, 1\}^P)^T$  is the reset function.

Let  $t$  be a transition, we denote  $\mathcal{R}(t)$  the vector defined by:

$$\forall p \in P, \mathcal{R}(t)[p] = \max(\blacklozenge(t)(p) * M(p), \bullet(t)(p))$$

A transition  $t_k$  is now said to be newly enabled by the firing of transition  $t_i$  from the marking  $M$  iff:

$$\uparrow \text{enabled}(t_k, M, t_i) = (\bullet(t_k) \leq M - \mathcal{R}(t_i) + (t_i)\bullet) \wedge [(t_k = t_i) \vee (\bullet(t_k) > M - \mathcal{R}(t_i))]$$

$\mathcal{R}(t)[p]$  represents the number of tokens to be removed from place  $p$  when firing transition  $t$ .

The new semantics is also defined as a TTS; only the discrete transition relation is affected.

**Definition 3.6** (Semantics of a TPN with reset arcs). *The semantics of a labeled time Petri net with reset arcs  $\mathcal{T}$  is defined as a TTS  $\|\mathcal{T}\| = (Q, q_0, \Lambda(T), \rightarrow)$  such that:*

- $Q \subseteq \mathbb{N}^P \times (\mathbb{R}_{\geq 0})^T$ ;
- $q_0 = (M_0, \bar{0})$ ;
- $\rightarrow \subseteq Q \times \{\Lambda(T) \cup \mathbb{R}_{\geq 0}\} \times Q$  is the transition relation including a discrete transition relation and a continuous transition relation.

- The discrete transition relation is defined by:

$$\forall t \in T \quad (M, v) \xrightarrow{\Lambda(t)} (M', v') \Leftrightarrow \begin{cases} t \in \text{enabled}(M) \\ a(t) \leq v(t) \leq b(t) \\ M' = M - \mathcal{R}(t) + (t)^\bullet \\ \forall t_k \in T, v'(t_k) = \begin{cases} 0 & \text{if } \uparrow \text{enabled}(t_k, M, t) \\ v(t_k) & \text{otherwise} \end{cases} \end{cases}$$

- The continuous transition relation is defined as above.

Let us consider the TPN illustrated Fig. 3.2. Firing transition  $T_1$  from the initial marking  $M_0 = \{3, 2, 1, 0\}$ , at any date between 2 and 5 time units (t.u.), leads to the marking  $\{0, 0, 2, 3\}$ .

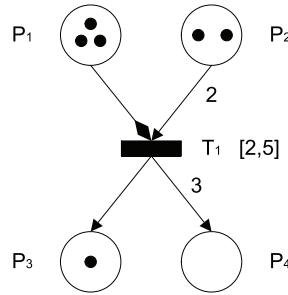


Figure 3.2: A simple time Petri net with a reset arc

## 3.2 Translation principles

The approach proposed in this work consists in performing a structural translation by using *elementary TPN patterns* for each type of node and for each item. A superscript is added to the place and transition labels to distinguish the elementary TPNs. The complete TPN is then simply obtained by connecting patterns together (the composition function is described in Section 3.4). Note that untimed EFFBDs are simply translated by Petri nets, times EFFBDs by TPNs and EFFBDs with forced terminations (with or without AA mode) by TPNs with reset arcs (more precisely, reset arcs are added at the composition but are not in the elementary patterns). Here, we directly present the most complex patterns.

### 3.2.1 Item pattern

An item  $A$  is simply translated by a single place  $pItem^A$  with the initial marking  $I_0(A)$ .

### 3.2.2 Control patterns

Each pattern encoding a node  $n$  is a labeled TPN  $\mathcal{T}^n = (P^n, T^n, \bullet(\cdot)^n, (\cdot)^\bullet^n, M_0^n, a^n, b^n, n, \Lambda^n)$ . Every node pattern is provided in Section 3.3. Each pattern begins with one entry place ( $m$  for an  $AND_{out}$  node with  $m$  predecessors or a function receiving  $m$  resources in the AA mode) and ends with one transition (2 for  $IT_{in}$  nodes,  $m$  for  $OR_{in}$  nodes and none for  $IT_{out}$  and  $LP_{out}$  nodes). The full descriptions of functions  $M_0^n, a^n$  etc. are given in Section 3.3.7.

## 3.3 Translation patterns

This section gives for each construct its EFFBD representation and the corresponding pattern alongside. Please note that since constructs usually consist of two nodes, most patterns are in fact composed of two elementary TPNs. When the interval associated to a transition is  $[0, 0]$ , it is omitted on the graphical representation of the TPN. Likewise, when the weight of an arc is 1, it is not shown on the arrow.

### 3.3.1 Parallel structures

Fig. 3.3 gives the pattern of a parallel structure without any kill branch. Note the  $pSynch^{x,o}$  places: they perform the synchronization ending the parallel structure ( $x$  being the name of the last construct on the parallel branch).

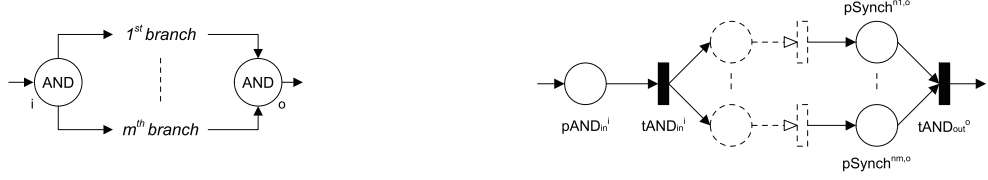


Figure 3.3: Patterns of nodes  $i \in AND_{in}$  and  $o \in AND_{out}$

Fig. 3.4 gives the pattern for a parallel structure with one kill branch (namely, the last one); the pattern is similar when more than one parallel branch hold the kill modifier but is not provided here, for readability reasons. In addition, only one “generic” reset arc is represented. It links every place corresponding to a construct contained in other parallel branches of the structure to transition  $tExit^{nm}$  (the exit transition of the pattern translating the last construct on the kill branch). Moreover, if any construct on the other branches has a pattern in which the initial marking of a place  $p$  is  $k > 0^1$ , then an arc of weight  $k$  is added, from  $tExit^{nm}$  to  $p$ , so as to reset the place to its initial marking (obviously, if the place corresponds to the initial node, it is not reset).

Finally, note the arc linking  $tExit^{nm}$  to the  $pSynch^{x,o}$  places: it forces all predecessors of the closing AND node to be in the *executed* state.

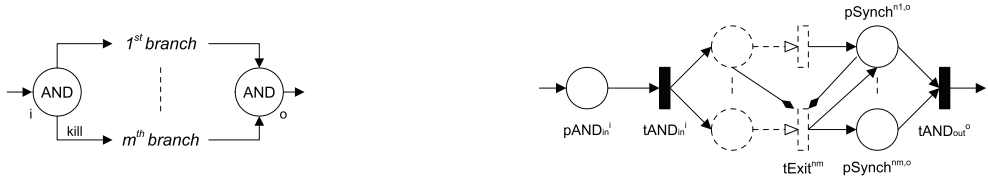


Figure 3.4: Patterns of nodes  $i \in AND_{in}$  and  $o \in AND_{out}$  with a kill branch

### 3.3.2 Selection structures

Fig. 3.5 gives the pattern of a selection structure.

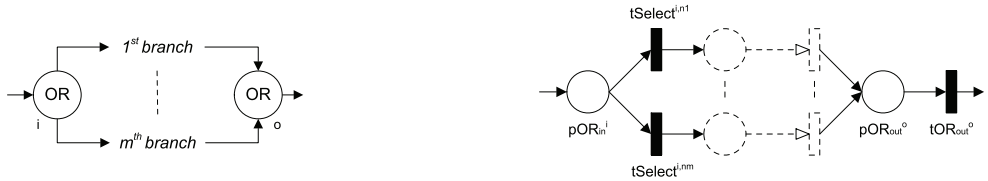


Figure 3.5: Patterns of nodes  $i \in OR_{in}$  and  $o \in OR_{out}$

### 3.3.3 Iteration structures

Fig. 3.6 gives the pattern of an iteration structure. Note the additional places  $pNoLess^a$  and  $pNoMore^a$ ; the former ensures that the iterate branch is taken *at least*  $i_M$  times while the latter ensures it is taken *at most*  $i_M$  times. The additional arc between  $pIT_{in}^i$  and  $tIT_{exit}^i$  enforces a complete reset of the pattern at the exit.

<sup>1</sup> See for instance the iteration pattern.

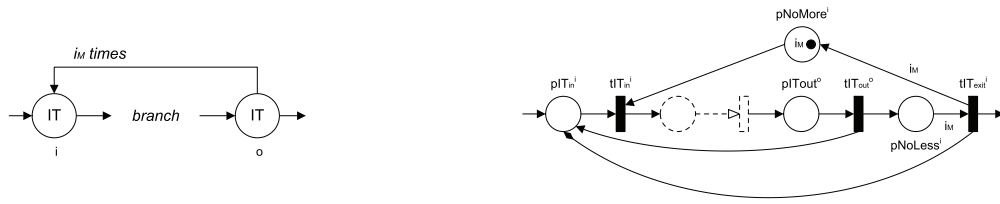


Figure 3.6: Patterns of nodes  $i \in IT_{in}$  and  $o \in IT_{out}$

### 3.3.4 Loop structures

Fig. 3.7 gives the pattern of a loop structure containing one LE node. Again, the pattern is extensible to the case of multiple LE nodes. As in the kill structure, reset arcs are added to empty any place created at the translation of the constructs contained in the loop branch. In addition (if needed), arcs link the transition  $tLE$  to any place in the loop branch whose initial marking is not 0.

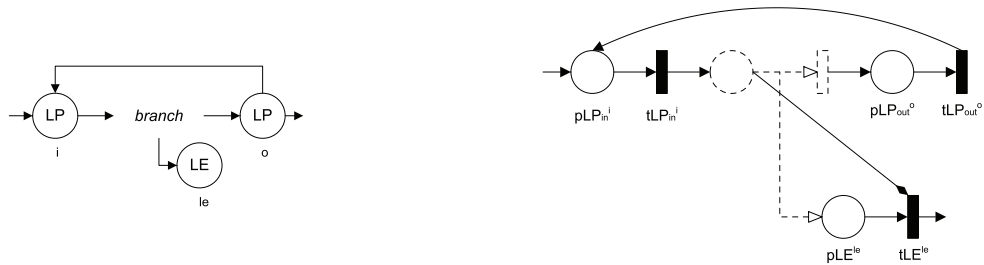


Figure 3.7: Patterns of nodes  $i \in LP_{in}$ ,  $o \in LP_{out}$  and  $le \in LE$

### 3.3.5 Non decomposed functions

The pattern adopted when a function consumes all its input resources in the non-AA mode is illustrated on a simple example, shown Fig. 3.8. Function  $F$  receives the resource  $rA$  in the quantity  $qA$  and the trigger  $tC$ . It produces  $qC$  units of resource  $rC$ .

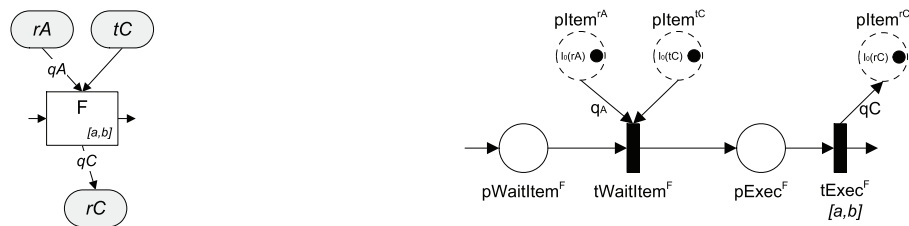
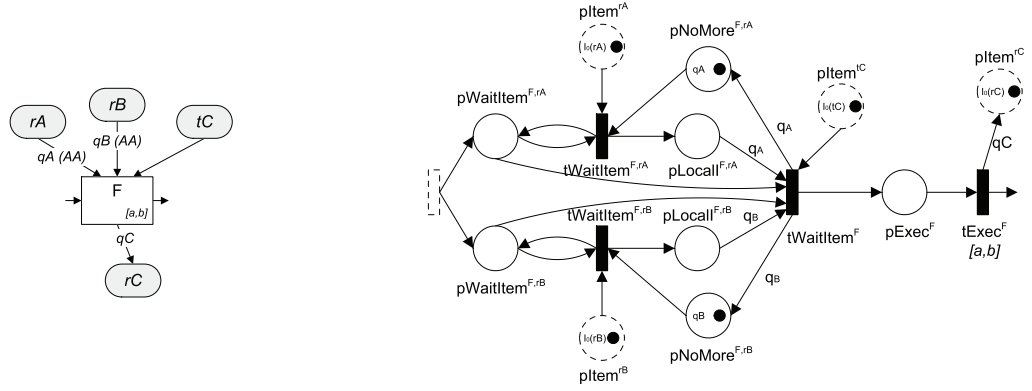


Figure 3.8: Patterns of function  $F$  (non-AA mode)

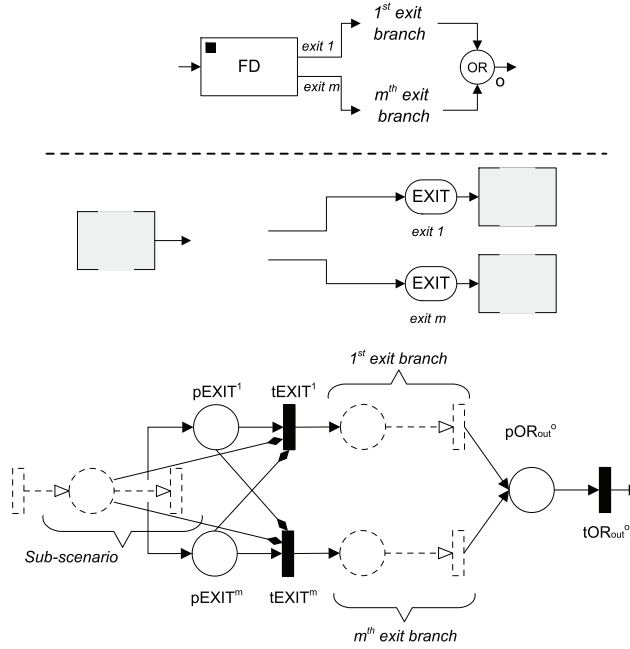
Fig. 3.9 provides the pattern used when the function consumes some of its input resources in the AA mode (here, both resources  $rA$  and  $rB$  are consumed in the AA mode). Note places  $pLocalI^{F,rX}$  and  $pNoMore^{F,rX}$ : they enforce the asynchronous consuming of  $rA$  and  $rB$ , in the correct quantities.

### 3.3.6 Sub-scenarios

Fig. 3.10 shows the pattern for a decomposed multi-exit function  $FD$ . Again, reset arcs are added to empty any place created at the translation of the constructs contained in the sub-scenario. In addition (if needed), weighted arcs link the transitions  $tEXIT^x$  to any place in the sub-scenario branch whose initial marking is not 0.

Figure 3.9: Patterns of function  $F$  (AA mode)

Note the reset arc between any  $pEXIT^i$  place and  $tEXIT^j$  transition ( $i \neq j$ ). They enforce the exit of the decomposed function through an unique exit branch.

Figure 3.10: Patterns of decomposed multi-exit function  $FD$ 

### 3.3.7 Node pattern summation

Table 3.1 recapitulates for each node type the entry place(s) and the exit transition(s). When obvious, the superscript  $n$  is omitted.

The initial marking  $M_0^n$  depends on the nature of the node. Table 3.2 gives for each concerned pattern the initial marking of the non-empty places.

Functions  $a^n$  and  $b^n$  also depend on the nature of  $n$ :

- if  $n \notin FC$ :  $\forall t \in T^n$ ,  $a^n(t) = b^n(t) = 0$
- if  $n \in FC$ :  $a^n(tExec^n) = \alpha(n)$  and  $b^n(tExec^n) = \beta(n)$ ; all remaining transition intervals are set to  $[0, 0]$

$n \in \dots$	$pEntry$	$tExit$
$AND_{in}$	$pANDin$	$tANDin$
$AND_{out}$	$\{pSynch^{X,n}\}$ with $X \in Pre(n)$	$tANDout$
$OR_{in}$	$pORin$	$\{tSelect^{n,X}\}$ with $X \in Post(n)$
$OR_{out}$	$PORout$	$tORout$
$IT_{in}$	$pITin$	$tITin, tITexit$
$IT_{out}$	$pITout$	–
$LP_{in}$	$pLPin$	$tLPin$
$LP_{out}$	$pLPout$	–
$LE$	$pLE$	$tLE$
$EXIT$	$pEXIT$	$tEXIT$
$FC$ s.t. $\forall res \in \mathcal{R}$ , $AA(n, res) = false$	$pWaitItem$	$tExec$
$FC$ s.t. $\exists res \in \mathcal{R}$ s.t. $AA(n, res) = true$	$\{pWaitItem^{n,A}\}$ with $A \in \mathcal{R}$ s.t. $AA(n, A) = true$	$tExec$

Table 3.1: Node entry places and exit transitions

$n \in \dots$	Place	Initial Marking	Remark
$AND_{in}$	$pANDin$	1	only if $n = n_0$
$OR_{in}$	$pORin$	1	
$IT_{in}$	$pITin$	1	
	$pNoMore$	$i_M$	where $i_M = count(n)$
$LP_{in}$	$pLPin$	1	only if $n = n_0$
$FC$ s.t. $\forall res \in \mathcal{R}$ , $AA(n, res) = false$	$pWaitItem$	1	
$FC$ s.t. $\exists res \in \mathcal{R}$ $AA(n, res) = true$	$\{pWaitItem^{n,A}\}$	1	if $n = n_0$ , where $A \in \mathcal{R}$ s.t. $AA(n, A) = true$
	$\{pNoMore^{n,A}\}$	k	where $k \in \mathbb{N}^*$ and $A \in \mathcal{R}$ s.t. $AA(n, A) = true$ and $(n, A, -k) \in \mathcal{A}_I$

Table 3.2: Initial marking of the patterns

Finally, the labeling function is defined as  $\forall t \in T^n \quad \Lambda^n(t) = n$ .

### 3.4 Construction of the complete net

Once every item and node pattern has been created, partial nets are connected with additional arcs to obtain the final net  $\mathcal{T} = (P, T, \bullet(\cdot), (\cdot)^\bullet, \blacklozenge(\cdot), M_0, a, b, \mathcal{N}, \Lambda)$

#### 3.4.1 Places and transitions

The sets  $P$  and  $T$  are defined as follows:

$$\begin{aligned} P &= (\cup_{n \in \mathcal{N}} P^n) \cup (\cup_{A \in \mathcal{I}} pItem^A) \\ T &= \cup_{n \in \mathcal{N}} T^n \end{aligned}$$

In the following, we adopt the notations:

- $P_{\mathcal{N}} = \cup_{n \in \mathcal{N}} P^n$  and  $P_{\mathcal{I}} = \cup_{A \in \mathcal{I}} pItem^A$
- $\forall t \in T$ ,  $\tau$  is the (unique) node such that  $t \in T^\tau$ ;
- $\forall p \in P_{\mathcal{N}}$ ,  $\pi$  is the (unique) node such that  $p \in P^\pi$ ;

- $\forall p \in P_{\mathcal{I}}, \iota$  is the (unique) item such that  $p = pItem^{\iota}$ .

The net initial marking is defined as:

$$\forall p \in P, M_0[p] = \begin{cases} M_0^{\pi} & \text{if } \exists \pi \in \mathcal{N} \text{ s.t. } p \in P^{\pi} \\ I_0(\iota) & \text{if } \exists \iota \in \mathcal{I} \text{ s.t. } p = pItem^{\iota} \end{cases}$$

$a$  is defined as  $\forall t \in T, a(t) = a^{\tau}(t)$ .  $b$  and  $\Lambda$  are likewise defined.

### 3.4.2 Additional arcs

The set of the TPN arcs is formed of:

- the “inner pattern arcs” (they are in  $\bullet(\cdot)$  and  $(\cdot)\bullet$ );
- connecting arcs between constructs (in  $(\cdot)\bullet$  only);
- “reset” arcs including the proper reset arcs (in  $\blacklozenge(\cdot)$ ) and “transition-to-place” arcs resetting the initially non-empty places (in  $(\cdot)\bullet$ );
- item consuming and producing arcs (in  $\bullet(\cdot)$  and  $(\cdot)\bullet$ ).

More precisely,  $\bullet(\cdot)$  is defined as:

$\forall t \in T$

$$\begin{aligned} \forall p \in P_{\mathcal{N}}, \bullet(t)[p] &= \begin{cases} \bullet(t)^{\tau}[p] & \text{if } \tau = \pi \\ 0 & \text{otherwise} \end{cases} \\ \forall p \in P_{\mathcal{I}}, \bullet(t)[p] &= \begin{cases} |k| & \text{if } \tau \in FC \wedge \wedge(\tau, \iota, -k) \in \mathcal{A}_I \wedge AA(\tau, \iota) = false \wedge t = tWaitItem^{\tau} \\ 1 & \text{if } \tau \in FC \wedge (\tau, \iota, -k) \in \mathcal{A}_I \wedge AA(\tau, \iota) = true \wedge t = tWaitItem^{\tau, \iota} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

where  $k \in \mathbb{N}^*$ .

Let  $t \in T$  and  $p \in P_{\mathcal{N}}$ ; the following expressions describe the conditions for the firing of  $t$  to “reset”  $p$  (i.e. when  $\tau$  terminates  $\pi$ ):

$$\begin{aligned} TermKill &: \begin{cases} Paths(\pi, \tau) = \emptyset \\ \exists \alpha_o \in AND_{out} \text{ s.t. } kill(\tau, \alpha_o) = true \\ t = tExit^{\tau} \text{ where } tExit^{\tau} \text{ is the exit transition in } T^{\tau} \\ \left\{ \begin{array}{l} \pi \in Cont(\langle \alpha_i, \alpha_o \rangle) \text{ where } \langle \alpha_i, \alpha_o \rangle \in AND \\ \text{or } p = pSynch^{n, \alpha_o} \text{ with } n \neq \tau \end{array} \right. \end{cases} \\ TermLE &: \begin{cases} Paths(\pi, \tau) = \emptyset \\ \tau \in LE \\ t = tLE^{\tau} \\ \pi \in Cont(Ref(\tau)) \end{cases} \\ TermEXIT &: \begin{cases} Paths(\pi, \tau) = \emptyset \\ \tau \in EXIT \\ t = tEXIT^{\tau} \\ \pi \in Cont(Ref(\tau)) \setminus \tau \end{cases} \end{aligned}$$

The preliminary condition  $Paths(\pi, \tau) = \emptyset$  means that the constructs on the same branch as  $\tau$  do not need to be reset (as they are necessarily inactive).

In addition, we define for each  $t \in T$  and each  $p \in P_{\mathcal{N}}$  the  $Connect(t, p)$  function that is true only if there is an arc going from  $t$  to  $p$  that connects to successive patterns:  $Connect(t, p) = true \Leftrightarrow$

- if  $t \in IT_{in}$ :
  - $t = tITin^\tau$  and  $p = pEntry^{\pi'}$  where  $\pi' \in Post(\tau)$
  - or  $t = tITexit^\tau$  and  $p = pEntry^{\pi''}$  where  $\pi'' \in Post(\bar{\tau})$  and  $\langle \tau, \bar{\tau} \rangle \in IT$
- if  $t \in IT_{out}$ :
  - $t = tITout^\tau$
  - and  $p = pITin^\tau$  or  $p = pNoLess^\tau$  where  $\langle \tau, \tau \rangle \in IT$
- if  $t \in LP_{out}$ :
  - $p = pLPout^\tau$  and  $p = pLPin^\tau$  where  $\langle \tau, \tau \rangle \in LP$
- if  $t \in LE$ :
  - $p = pLE^\tau$  and  $p = pEntry^{\pi'}$  where  $\pi' \in Post(lp_{out})$  and  $\langle lp_{in}, lp_{out} \rangle \in LP$
- otherwise:
  - $t = tExit^\tau$
  - and  $p = pEntry^\pi$
  - and  $(\tau, \pi) \in \mathcal{A}_I$

$(.)^\bullet$  is then defined by:

$\forall t \in T$

$$\forall p \in P_{\mathcal{N}}, \quad \bullet(t)[p] = \begin{cases} \bullet(t)^\tau[p] & \text{if } \tau = \pi \\ 1 & \text{if } Connect(t, p) \\ M_0^\pi[p] & \text{if } (TermKill \vee TermLE \vee TermEXIT) \wedge \pi \neq n_0 \\ 0 & \text{otherwise} \end{cases}$$

$$\forall p \in P_{\mathcal{I}}, \quad \bullet(t)[p] = \begin{cases} k & \text{if } \tau \in FC \wedge t = tExec^\tau \wedge (\tau, \iota, k) \in \mathcal{A}_I \\ 0 & \text{otherwise} \end{cases}$$

where  $k \in \mathbb{N}^*$ .

Note that the different conditions are mutually exclusive.

Finally,  $\blacklozenge(.)$  is defined by:

$\forall t \in T$

$$\forall p \in P_{\mathcal{N}}, \quad \blacklozenge(t)[p] = \begin{cases} 1 & \text{if } (TermKill \vee TermLE \vee TermEXIT) \\ 0 & \text{otherwise} \end{cases}$$

$$\forall p \in P_{\mathcal{I}}, \quad \blacklozenge(t)[p] = 0$$

# Chapter 4

## Properties and results

This chapter presents some properties of the EFFBDs and the TPNs obtained by application of the translation patterns described in the preceding chapter. In addition, after proving the behavioral equivalence between both models, some interesting results on TPNs are applied to EFFBDs.

### 4.1 Non-reentrance and boundedness of the EFFBDs

A fundamental result on well-formed EFFBDs is provided below.

**Proposition 4.1** (Non re-entrance of the EFFBDs). *A well-formed EFFBD is not reentrant i.e. each node and structure must be exited before being enabled again.*

*Proof.* As there is only one initial node in the EFFBD, only opening AND nodes can create two (or more) independent control flows. However, under the assumption of having a well-formed EFFBD, these flows cannot converge to any node but the corresponding closing AND node. Therefore, no node can be enabled while still in execution.  $\square$

In the following, EFFBDs are supposed to be well-formed.

A number of powerful results have been proved for *bounded* TPNs (i.e. for which the marking of any place stays finite). Likewise, this section provides a few results on a sub-class<sup>1</sup> of EFFBDs so-called bounded EFFBDs and whose definition is given below.

**Definition 4.1** (Bounded EFFBD). *Let  $\mathcal{E}$  be an EFFBD and  $(S, s_0, \mathcal{N}, \rightarrow)$  its semantics.  $\mathcal{E}$  is bounded iff:*

$$\forall(A, C, I, localI, \nu) \in S, \forall item \in \mathcal{I}, \exists k \in \mathbb{N} \quad I(item) \leq k$$

Two sufficient conditions to ensure the boundedness of an EFFBD are given hereunder. The proofs are trivial enough to be omitted (they rely on the fact that the only potentially unbounded places correspond to items and that the only “infinite behavior” is caused by loops).

**Proposition 4.2.** *An EFFBD that contains no item is bounded.*

**Proposition 4.3.** *An EFFBD in which no loop construct contains item producing functions is bounded.*

An example of such bounded EFFBD is given hereunder. The EFFBD shown Fig. 4.1 represents a very basic system where a task `Write` writes data in a bounded buffer (initially empty), a task `Read` reads (and erases) the data in the buffer and a monitoring task, `Terminate`, can end the activity of the system. Items `Buffer`, initialized to 0, and `InvBuffer` represent the filling state of the buffer; the initial amount of `InvBuffer` is the size of the buffer.

---

<sup>1</sup> Most of the models designed in SE are actually bounded; therefore, the restriction is not too important.

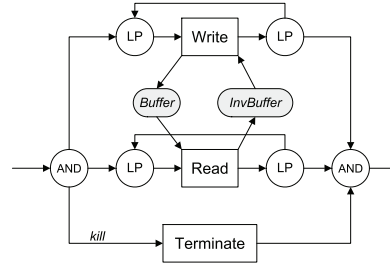


Figure 4.1: Example of a bounded EFFBD (model of a bounded buffer)

In addition, to write data in a full buffer, **Write** must wait for **Read** to free at least one space; conversely, **Read** cannot access an empty buffer and must wait for **Write** to provide at least one data. Here, the actual data is not represented, no more than the mechanism determining which data to read from the buffer. Moreover, no protection against simultaneous access is defined here: the purpose of this example is simply to show that bounded EFFBDs are not too restrictive.

The consequence on the boundedness of the resulting TPN is immediate.

**Proposition 4.4.** *Let  $\mathcal{E}$  be a bounded EFFBD,  $(S, s_0, \mathcal{N}, \rightarrow_{\mathcal{E}})$  its semantics and  $m$  defined as:*

$$m = \max\left(\max_{iter \in IT_{in}} count(iter), maxI, \max_{(f, A, k) \in \mathcal{A}_I, k < 0} |k|\right)$$

where:

$$maxI = \max_{(A, C, I, localI, \nu) \in S} \left( \max_{item \in \mathcal{I}} I(item) \right)$$

Let  $\mathcal{T}$  be the TPN obtained from  $\mathcal{E}$  and  $(Q, q_0, T, \rightarrow)$  its semantics.  $\mathcal{T}$  is  $m$ -bounded:

$$\forall (M, v) \in Q, \forall p \in P \quad M(p) \leq m$$

*Proof.* As there is only one initial node, each node pattern is, by construction, 1-bounded (or *safe*) except for  $IT_{in}^n$  patterns, which are  $k$ -bounded with  $k = count(n)$ , and  $FC$  patterns (in the AA mode) which are also  $k$ -bounded with  $k$  being the highest quantity needed for an AA-mode consumed resource. In addition, as the EFFBD is bounded, all  $pItem^X$  places in  $\mathcal{T}$  keep a finite marking. Therefore,  $\mathcal{T}$  is bounded.  $\square$

## 4.2 Strong timed bisimulation

A binary relation  $\sim$  is defined over the behavior of EFFBD models and the corresponding TPNs.

In order to prove the correctness of the method exposed above and to extend known results from TPNs to EFFBDs, the equivalence between both models should be proved. It is thus necessary to define first a binary relation  $\sim$  over the behavior of EFFBD models and their corresponding TPNs.

Let  $\mathcal{E} = (\mathcal{N}, \mathcal{I}, \mathcal{A}, count, n_0, I_0, a, b, kill, AA)$  be an EFFBD,  $\mathcal{T}_{\mathcal{E}} = (P, T, \bullet(\cdot), (\cdot)^{\bullet}, \blacklozenge(\cdot), M_0, \alpha, \beta, \mathcal{N}, \Lambda)$  the labeled TPN obtained by the translation of  $\mathcal{E}$ . Let  $\|\mathcal{E}\| = (S, s_0, \mathcal{N}, \rightarrow_{\mathcal{E}})$  and  $\|\mathcal{T}_{\mathcal{E}}\| = (Q, q_0, \mathcal{N}, \rightarrow_{\mathcal{T}})$  be their respective semantics. The binary relation  $\sim \subseteq S \times Q$  is defined as follows:

$$\forall s = (A, C, I, localI, \nu) \in S, \forall q = (M, v) \in Q$$

$$s \sim q \Leftrightarrow \begin{cases} Activities_{enabled} \wedge Activities_{executing} \wedge Activities_{executed} \\ Counters \\ Items \\ LocalItems \\ Valuations \end{cases}$$

with:

$$\begin{aligned}
\text{Activities}_{\text{enabled}} &: \forall n \in \mathcal{N} : A(n) = \text{enabled} \Leftrightarrow \left\{ \begin{array}{l} M(\text{pEntry})^n = 1 \text{ if } n \notin \text{AND}_{\text{out}} \cup \text{FC} \\ \sum_{n' \in \text{Pre}(n)} M(\text{pSynch}^{n',n}) \geq 1 \text{ if } n \in \text{AND}_{\text{out}} \\ M(\text{pWaitItem}) = 1 \text{ if } n \in \text{FC} \\ \wedge \forall \text{res} \in \mathcal{R}, \text{AA}(n, \text{res}) = \text{false} \\ \forall \text{res} \in \mathcal{R} / \text{AA}(n, \text{res}), \text{pWaitItem}^{n, \text{res}} = 1 \\ \vee \text{pLocalI}^{n, \text{res}} \geq 1 \text{ otherwise} \end{array} \right. \\
\text{Activities}_{\text{executing}} &: \forall f \in \text{FC} : A(f) = \text{executing} \Leftrightarrow M(\text{pExec}^f) = 1 \\
\text{Activities}_{\text{executed}} &: \forall n \in \mathcal{N} \text{ s.t. } \exists \alpha_o \in \text{AND}_{\text{out}} \cap \text{Post}(n) : A(n) = \text{executed} \Leftrightarrow M(\text{pSynch}^{n, \alpha_o}) = 1 \\
\text{Counters} &: \forall \text{it} \in \text{IT}_{\text{in}}, \forall k \in \mathbb{N} : C(\text{it}) = k \Leftrightarrow M(\text{pNoMore}^{\text{it}}) = \text{count}(\text{it}) - k \\
\text{Items} &: \forall A \in \mathcal{I}, \forall k \in \mathbb{N} : I(A) = k \Leftrightarrow M(\text{pItem}^A) = k \\
\text{LocalItems} &: \forall A \in \mathcal{R}, \forall f \in \text{FC} / \text{AA}(f, A), \forall k \in \mathbb{N} : \text{localI}(f, A) = k \Leftrightarrow M(\text{pLocal}^{f, A}) = k \\
\text{Valuations} &: \forall f \in \text{FC}, \forall x \in \mathbb{R}_{\geq 0} : \nu(f) = x \Leftrightarrow v(\text{tExec}^f) = x
\end{aligned}$$

**Proposition 4.5.** *The relation  $\sim$  is a strong timed bisimulation relation.*

The proof is given hereunder.

*Proof.* The general principle is to prove each point of definition 2.2. Only well-formed EFFBDs are considered here. The proof mechanism is based on a structural induction: the bisimulation is proved, point by point, one every transition possible (here, only the  $\text{AND}_{\text{in}}$  case is given; the proof for other nodes follows the same arguments and is left to the reader). Due to the atomicity, imbrication and continuity properties of the well-formed EFFBD, these elementary bisimulation results are straightforwardly propagated to any combination.

Thus, points  $R_{\text{action}}$  and  $R'_{\text{action}}$  shall only be proved in the case transition  $a$  is in  $\text{AND}_{\text{in}}$ . The demonstration for other nodes follows the same arguments and is left to the reader.

Let  $\mathcal{E}_{\text{AA}} = (\mathcal{N}, \mathcal{I}, \mathcal{A}, \text{count}, n_0, I_0, a, b, \text{kill}, \text{AA})$  be an EFFBD and  $\mathcal{T}_{\mathcal{E}} = (P, T, \bullet(\cdot), (\cdot)\bullet, \blacklozenge(\cdot), M_0, \alpha, \beta, \mathcal{N}, \Lambda)$  the TPN obtained by the translation of  $\mathcal{E}$ . Let  $\|\mathcal{E}\| = (S, s_0, \mathcal{N}, \rightarrow_{\mathcal{E}})$  and  $\|\mathcal{T}_{\mathcal{E}}\| = (Q, q_0, T, \rightarrow_{\mathcal{T}})$  be their respective semantics and  $\sim$  be the binary relation defined as in Section 4.2.

$R_0$  Trivially,  $s_0 \sim q_0$

$R_d$  Let  $s = (A, C, I, \text{localI}, \nu)$ ,  $s' = (A, C, I, \text{localI}, \nu') \in S$  and  $d \in \mathbb{R}_{>0}^1$  s.t.  $s \xrightarrow{d}_{\mathcal{E}} s'$ . Let  $q = (M, v) \in Q$  s.t.  $s \sim q$ . It follows:

- from the definition of  $\|\mathcal{E}\|$ ,  $\nu' = \nu + d$  and  $\forall f \in \text{FC}$  s.t.  $A(f) = \text{executing}$ ,  $\nu(f) + d \leq b(f)$
- according to the definition of  $\sim$  and to the pattern of a function,  $\forall f \in \text{FC}$ ,  $\nu(f) = v(\text{tExec}^f)$  and  $b(f) = \beta(\text{tExec}^f)$
- $\text{enabled}(M) = \{\text{tExec}^f \mid A(f) = \text{executing}\}$  as nodes  $n \notin \text{FC}$  are not enabled and  $\forall f' \in \text{FC}$  s.t.  $A(f') = \text{enabled}$ , some resource is still missing i.e.  $\text{tWaitItem}^{f'} \notin \text{enabled}(M)$  in the AA mode and, in the AA mode, all the  $\text{tWaitItem}^{f', A}$  transitions (with  $A$  such that  $\text{AA}(f', A) = \text{true}$ ) as well as  $\text{tSynch}^{f'}$  are not in  $\text{enabled}(M)$
- as a result,  $\forall t \in \text{enabled}(M)$ ,  $v(t) + d \leq \beta(t)$

Let  $q' = (M, v + d)$ . Trivially,  $q \xrightarrow{d}_{\mathcal{T}} q'$  and  $s' \sim q'$ .

$R'_d$  Conversely, let  $q = (M, v)$ ,  $q' = (M, v') \in Q$  and  $d \in \mathbb{R}_{>0}$  s.t.  $q \xrightarrow{d}_{\mathcal{T}} q'$ . Let  $s = (A, C, I, \text{localI}, \nu) \in S$  s.t.  $s \sim q$ . It follows:

---

<sup>1</sup> If  $d = 0$ , the case is trivial.

- from the definition of  $\|\mathcal{T}_\mathcal{E}\|$ ,  $v' = v + d$  and  $\forall t \in \text{enabled}(M)$ ,  $v(t) + d \leq \beta(t)$
- according to the definition of  $\sim$  and to the pattern of a function,  $\forall f \in FC$ ,  $\nu(f) = v(tExec^f)$  and  $b(f) = \beta(tExec^f)$
- since  $d > 0$ , only  $tExec^f$  transitions are enabled where  $f \in FC$  i.e.  $\forall n \notin FC$ ,  $A(n) \neq \text{enabled}$  and  $\forall f' \in FC$ ,  $A(f') = \text{enabled} \Rightarrow$  some items are missing
- as a result,  $\forall f \in FC$ ,  $A(f) = \text{executing} \Rightarrow \nu(f) + d \leq b(f)$

Let  $s' = (A, C, I, localI, \nu + d)$ . Trivially,  $s \xrightarrow{d}_\mathcal{E} s'$  and  $s' \sim q'$ .

$R_\sigma$  Let  $s = (A, C, I, localI, \nu)$ ,  $s' = (A', C, I, localI, \nu) \in S$  and  $\sigma \in AND_{in}$  s.t.  $s \xrightarrow{\sigma}_\mathcal{E} s'$ . Let  $q = (M, v) \in Q$  s.t.  $s \sim q$ . It follows:

- by definition of  $\|\mathcal{E}\|$ ,  $A(\sigma) = \text{enabled}$ , any node contained in the AND structure is inactive,  $A'(\sigma) = \text{inactive}^1$  and  $\forall n \in Post(\sigma)$ ,  $A'(n) = \text{enabled}$
- according to the definition of  $\sim$  and to the translation patterns,  $M(pANDin^\sigma) = 1$  and therefore  $tANDin^\sigma \in \text{enabled}(M)$
- $v(tANDin^\sigma) = 0$

Let  $q' = (M', v)$  s.t.  $M'(pANDin^\sigma) = 0$  and  $\forall n \in Post(\sigma)$ ,  $M'(pEntry^n) = 1$  where  $pEntry^n$  is the entry place of the TPN  $\mathcal{T}^n$ . Trivially,  $q \xrightarrow{\sigma}_\mathcal{T} q'$ . According to the translation patterns, the firing interval of any transition newly enabled by  $M'$  is  $[0, 0]$  and therefore  $s' \sim q'$ .

$R'_\sigma$  Conversely, let  $q = (M, v)$ ,  $q' = (M, v') \in Q$ ,  $\sigma \in AND_{in}$  s.t.  $q \xrightarrow{\sigma}_\mathcal{T} q'$ . Let  $s = (A, C, I, localI, \nu) \in S$  s.t.  $s \sim q$ . It follows:

- by definition of  $\|\mathcal{T}_\mathcal{E}\|$ ,  $M(pANDin^\sigma) = 1^2$  and  $\forall p$  s.t.  $(tANDin^\sigma)^\bullet[p] = 1$ ,  $M'(p) = 1$
- according to the definition of  $\sim$  and to the translation patterns,  $A(\sigma) = \text{enabled}$ ,  $A'(\sigma) = \text{inactive}$  and  $\forall n \in Post(\sigma)$ ,  $A(n) = \text{enabled}$

Let  $s' = (A', I, C, localI, \nu)$  s.t.  $A'(\sigma) = \text{inactive}$  and  $\forall n \in Post(\sigma)$ ,  $A'(n) = \text{enabled}$ . Trivially,  $s \xrightarrow{\sigma}_\mathcal{E} s'$ . Finally, according to the translation patterns, the valuation of any  $tExec^f$  transition for  $f \in FC$  is not affected by the firing of  $tANDin^\sigma$ . Therefore  $s' \sim q'$ .

Since nodes are either in sequence or fully nested, the bisimulation relation is propagated throughout the complete EFFBD. The binary relation  $\sim$  is therefore a strong timed bisimulation. □

### 4.3 Additional results

**Theorem 4.6** (Decidability of the  $k$ -boundedness). *For any TPN  $N$  with the semantics  $(Q, q_0, T, \rightarrow)$  and a given  $k \in \mathbb{N}$ , the following problem is decidable [BM83]:*

$$\forall (M, v) \in Q, \quad \forall p \in P : \quad M(p) \leq k \quad ?$$

Since reset arcs do not increase the marking of the net, this problem is also decidable for TPNs with reset arcs. More generally, all following results are extendable to TPNs with reset arcs and EFFBDs with forced terminations and AA mode.

**Corollary 4.1.** *For any EFFBD  $\mathcal{E}$  with the semantics  $(S, s_0, \mathcal{N}, \rightarrow)$  and a given  $k \in \mathbb{N}$ , the following problem is decidable:*

$$\forall (A, C, I, localI, \nu) \in S, \forall A \in \mathcal{I} : \quad I(A) \leq k \quad ?$$

*Proof.* Using proposition 4.5 and theorem 4.6, the proof is immediate. □

<sup>1</sup>  $A'(\sigma) \neq \text{executed}$  in a EFFBD where no branch is empty, which is supposed true here.

<sup>2</sup>  $M(pANDin^\sigma) \leq 1$  as shown in Section 4.1.

As a result, it is always possible to check whether the item level of any EFFBD stays under a limit specified by the system designer, which is particularly needed when assessing the size of a system in the design process.

**Theorem 4.7.** *For any bounded TPN  $\mathcal{T}$  with the semantics  $(Q, q_0, T, \rightarrow)$ , the following problems are decidable [BM83]:*

- *Accessibility of a marking:* “given a marking  $M$ , is there a state  $(M', v') \in Q$  such that  $M = M'$ ?”
- *Accessibility of a state:* “given a state  $q = (M, v)$ ,  $q \in Q$ ?”
- *Liveness:* “given a state  $q \in Q$  and a transition  $t \in T$ , are there a state  $q' \in Q$  and a transition sequence ending with  $t$  from  $q$  to  $q'$ ?”

**Corollary 4.2.** *For any item-bounded EFFBD  $\mathcal{E}$  with the semantics  $(S, s_0, \mathcal{N}, \rightarrow)$  the following problems are decidable:*

- *Accessibility of an activity state:* “given a node  $n$ , can  $n$  be enabled ?”
- *Accessibility of a state:* “given a state  $s$ ,  $s \in S$ ?”

Moreover, as mentioned in the introduction, the final purpose of this work is to assess the safety of the designed systems. It is therefore necessary to express sometime complex properties such as “*Upon the reception of an alarm, the system always reacts in an appropriate way in less than 5 time units.*”

In that respect, a temporal logic such as the Time Computation Tree Logic (TCTL [ACD90]) is particularly well adapted to express these specifications. It should be noted that the model checking of TCTL has been proved decidable on bounded TPN [CR06]. Moreover, a subset of TCTL, named TPN-TCTL, has been described in [Gar06] where, informally speaking, the atomic propositions of the formula are expressed in terms of linear inequalities on the marking of the TPN. The decidability of TPN-TCTL on bounded TPN has also been proved as decidable. In addition, a TPN-TCTL model checker was implemented in the ROMÉO software, a TPN analysis tool developed by the authors of [GLMR05]. As a result, if a property over an item-bounded EFFBD can be translated into a TPN-TCTL formula over the corresponding TPN, then it is also decidable.

# Conclusion and further work

This technical report has proposed a formal description and the behavioral semantics of EFFBDs, including some of the main variants. To our knowledge, it had never been formally expressed, although the language is widely used in SE design. This first step led to the definition of a transformation method from EFFBDs to TPNs, proved as preserving the temporal behavior of the high-level model. As a result, a number of fundamental properties, inherited from the research works carried on TPN were applied to EFFBDs. Although not discussed here, this work indicates the possibility and benefit of performing safety assessment on models designed by a typical systems engineer via model-checking techniques, in a completely transparent way.

Further work should focus on proving the correctness of the translation of high-level properties to TPN-TCTL formulas, and on the study of the complexity of the method algorithms. We also plan to describe the tools developed in application of those results in a further report. In addition, it is planned to extend the original EFFBD formalism to model “hybrid entities” such as continuous rates for the production or consuming of items etc.

# Bibliography

- [ACD90] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *5<sup>th</sup> IEEE Symposium on Logic in Computer*, pages 414–425, June 1990.
- [BCH<sup>+</sup>05] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux. Comparison of different semantics for time Petri nets. In *Automated Technology for Verification and Analysis (ATVA '05)*, volume 3707 of *Lecture Notes in Computer Science*. Springer, October 2005.
- [BM83] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. In *IFIP Congress Series*, volume 9, pages 41–46, 1983.
- [BMIS04] Simonetta Balsamo, Antiniscia Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, 2004.
- [CR06] F. Cassez and O. H. Roux. Structural translation from time Petri nets to timed automata – model-checking time Petri nets via timed automata. *The Journal of Systems and Software*, 79(10):1456–1468, 2006.
- [DFS98] C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *ICALP'98 – Lecture Notes in Computer Science*, volume 1443, pages 103–115, 1998.
- [Gar06] G. Gardey. *Contribution à la vérification et au contrôle des systèmes temps réel – Application aux réseaux de Petri temporels et aux automates temporisés*. PhD thesis, École Centrale de Nantes – Université de Nantes, December 2006.
- [GLMR05] G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Romeo: A tool for analyzing time Petri nets. In *Proceedings of the 17<sup>th</sup> International Conference on Computer-Aided Verification (CAV'05)*, Lecture Notes in Computer Science, 2005.
- [Her04] E. Herzog. *An approach to Systems Engineering tools data representation and exchange*. PhD thesis, Linköpings Universitet, 2004.
- [HMP92] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. *Real Time: Theory in Practice – Lecture Notes in Computer Science*, 1992.
- [IEE94] IEEE. IEEE-1220: Application & management of systems engineering process, 1994.
- [INC04a] INCOSE. *What is Systems Engineering?*, 2004. Available online: <http://www.incose.org>.
- [INC04b] INCOSE Technical Board. *Systems Engineering handbook: a “what to” guide for all SE practitioners*. INCOSE, 2a edition, June 2004.
- [Lon95] J. Long. Relationships between common graphical representations in Systems Engineering. In *5<sup>th</sup> International Symposium of the INCOSE*, St. Louis (MO), USA, July 1995. Updated July 2002.
- [Mag07] M. Magnin. *Réseaux de Petri à chronomètres – Temps continu et temps discret*. PhD thesis, École Centrale de Nantes – Université de Nantes, December 2007.

- [Mer74] P. Merlin. *A study of recoverability of computing systems*. PhD thesis, Dpt. of Computer Science, University of California, Irvine (CA), USA, 1974.
- [Par81] D.M. Park. Concurrency on automata and infinite sequences. *Conf. on Theoretical Computer Science*, Lecture Notes in Computer Science, 104, 1981.
- [Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, Germany, 1962.
- [Rug05] A.-E. Rugina. System dependability evaluation using AADL (Architecture Analysis and Design Language). In *Rencontres Jeunes Chercheurs en Informatique Temps-Réel (RJCITR)*, September 2005.
- [SAE04] SAE Aerospace. Architecture Analysis & Design Language (AADL) – SAE AS 5506, November 2004. Available online: <http://www.aadl.info>.
- [SLR07] C. Seidner, J.-P. Lerat, and O. H. Roux. Usability of formal verification on EFFBD models: Applying Petri nets to Systems Engineering issues. In *17<sup>th</sup> International Symposium of the International Council on Systems Engineering (IS2007)*, San Diego, CA, June 2007.
- [Sys07] SysML Finalization Task Force. *OMG Systems Modeling Language (OMG SysML<sup>TM</sup>) Specification – Proposed Available Specification*. Object Management Group, March 2007. Available online: <http://www.omg.org/cgi-bin/apps/doc?ptc/07-02-03.pdf>.
- [US 68] US Air Force. MIL-STD-499 Functional Flow Diagrams, 1968. DI-S-3604/S-126-1.
- [VD05] Y. Vandeperren and W. Dehaene. SysML and Systems Engineering applied to UML-based SoC design. In *2<sup>nd</sup> UML-SoC Workshop at 4<sup>2<sup>nd</sup></sup> Design Automation Conference*, Anaheim (CA), USA, June 2005.
- [Vit00] Vitech Corporation. *COREsim User Guide*. Vienna (VA), USA, third edition, 2000.