ϵ-TPN: definition of a Time Petri Net formalism simulating the behaviour of the timed grafcets

Médésu Sogbohossou, Antoine Vianou

ε-TPN: definition of a Time Petri Net formalism simulating the behaviour of the timed grafcets

Médésu Sogbohossou — Antoine Vianou

Département Génie Informatique et Télécommunications
École Polytechnique d'Abomey-Calavi (EPAC), 01 BP 2009 Cotonou, BENIN
{medesu.sogbohossou,antoine.vianou}@epac.uac.bj

ABSTRACT. To allow a formal verification of timed GRAFCET models, many authors proposed to translate them into formal and well-reputed languages such as timed automata or Time Petri nets (TPN). Thus, the work presented in [Sogbohossou, Vianou, Formal modeling of grafcets with Time Petri nets, IEEE Transactions on Control Systems Technology, 23(5)(2015)] concerns the TPN formalism: the resulting TPN of the translation, called here ε-TPN, integrates some infinitesimal delays (ε) to simulate the synchronous semantics of the grafcet. The first goal of this paper is to specify a formal operational semantics for an ε-TPN to amend the previous one: especially, priority is introduced here between two defined categories of the ε-TPN transitions, in order to respect strictly the synchronous hypothesis. The second goal is to provide how to build the finite state space abstraction resulting from the new definitions.

RÉSUMÉ. Afin de permettre la vérification formelle des grafcets temporisés, plusieurs auteurs ont proposé de les traduire dans des langages formels de réputation tels que les automates temporisés et les réseaux de Petri temporels (TPN). Ainsi, les travaux présentés dans [Sogbohossou, Vianou, Formal modeling of grafcets with Time Petri nets, IEEE Transactions on Control Systems Technology, 23(5)(2015)] concernent le formalisme des TPN: le réseau résultant de la traduction, dénommé ici ε-TPN, intègre des délais infinitésimaux (ε) pour simuler la sémantique synchrone du grafcet. Le premier objectif de cet article est de définir la sémantique opérationnelle d'un ε-TPN afin d'améliorer l'ancienne définition: spécifiquement, une priorité est introduite ici entre deux catégories de transitions définies pour ces réseaux, dans l'optique de respecter rigoureusement l'hypothèse synchrone. Le second but est de fournir une méthode de calcul de l'espace d'état fini qui découle des nouvelles définitions.

KEYWORDS : Time Petri Net, timed grafcet, state class, partial order execution, synchronous modelling

MOTS-CLÉS : Réseau de Petri temporel, grafcet temporisé, classe d'état, exécution ordre partiel, modélisation synchrone

1. Introduction

Formal specification of a critical system at the early stage of conception is often needed to achieve their reliability in working, by means of languages allowing simulation or formal verification on the established model of this system [8]. Graphical state-transition modeling formalisms in engineering are appreciated because of their intuitiveness. They are based on the automata theory, ensuring an unambiguous description of the behaviours of a system. Petri nets (PN) are one of these formalisms, used to model in a compact and explicit way the concurrency and the synchronization between the dynamic components of the so-called discrete-event systems [7]. In PNs, firing of transitions (with possibly multiple concurrent firings in the same instant) changes the state and express the dynamics of the modeled system. Time Petri nets (TPN) [2] are one of its extensions, suitable when quantitative time analyses are required for the real-time specifications.

Otherwise, the engineering pratices often promote less formal graphical languages, because of their increased semantic richness (for instance, litteral formulae and hierarchical modeling do not exist in the ordinary PNs) favoring more compact and fluent modeling to the detriment of unambiguous interpretations. These are the cases of formalisms derived from PNs, such as GRAFCET [1] (IEC 60848 standard) [9] and SFC (Sequential Function Chart, IEC 61131-3 standard) [10], used mainly in the world of the manufacturing control. Whereas simultaneous fireable transitions are always done by their total interleaving with PNs, the semantics of these two IEC standards considers only synchronous firings; a consequence is that the notion of transitions in conflict does not exist in GRAFCET and SFC formalisms. GRAFCET is intended for specification purposes (event-driven modeling), contrary to SFC for implementation uses (clock-driven modeling), and is considered in the sequel.

To allow a formal verification of GRAFCET (or SFC) models integrating quantitative time informations, many authors proposed to translate them into formal and well-reputed languages such as timed automata [11] or TPN [13, 12]. The work in [12] is focused on defining some transformation rules which are used to translate the entities composing a timed and not necessarily sound grafcet chart (steps, transitions, literal variables, actions) into connected blocks to obtain the resulting TPN. The method exploits the similarity between TPN and GRAFCET to avoid exponential size of the translation, and implicitly relies on a clear choice about the GRAFCET semantics.

To deal with synchronous firings inherent to GRAFCET formalism, the authors [12] introduced transitions with infinitesimal ε delays, however without redefining formally the resulting extended TPN. The first goal of this paper is to palliate this lack, by specifying a formal semantics for the so-called ε-TPN; the slight differences with the definition in [12] are also presented. Basing on this new definition, the second goal is to provide how to build the state space abstraction of an ε-TPN (which is just sketched in [12]) with the ε delays. Particularly, it is shown how to take advantage from this kind of TPN to cope with the state-space explosion problem, by avoiding useless interleaving of concurrent firings and by abstracting some state classes during the state-space construction.

In the next section are mainly recalled definitions about TPNs and GRAFCET charts. Section 3 summarizes and illustrates how the entities of a grafcet are converted into the connected subnets forming the corresponding TPN; also, a rule of translation is enhanced (concerning stored action modelling), and various definitions used in the subsequent sec-

1. Acronym in French: *GRAphe Fonctionnel de Commande Etape Transition*.

tions are introduced. Formal definitions about the syntax and semantics of ε-TPN are given in Section 4, completed with a characterisation of the ε-TPN model in the form of a list of features used in the next section throughout the proposal of state-space abstractions. Then, Section 5 extends the definition of a TPN state class and justifies the partial-order approach to compute a ε-TPN state-space abstraction, before presenting the algorithm of the all situations state-space and the stable situations state-space. Finally, this paper contribution is summarized and some perspectives are sketched in Section 6.

2. Definitions: time Petri nets and GRAFCET charts

In this section is recalled the syntax and semantics definitions about (classic) time Petri nets (TPN). Then, GRAFCET charts are presented and the related semantics choices are remembered, concerning especially the synchronous assumptions on which the subsequent sections are relied on.

2.1. Time Petri nets (TPN) and state-space abstractions

2.1.1. Syntax

Structurally, a time Petri net is defined as follows:

Definition 1. *A* Time Petri net *(TPN) is a tuple* $(P, T, W, W_I, W_R, ED, LD, M_0)$ *such as:*

1) *the nodes:* P *is the set of* places *and* T *is the set of* transitions *(*$P \cap T = \emptyset$*);*
2) *the* regular arcs *and the corresponding weights between nodes,* $W : P \times T \cup T \times P \longrightarrow \mathbb{N}$*;*
3) *the* read arcs*,* $W_R : P \times T \longrightarrow \mathbb{N}$*;*
4) *the* inhibitor arcs*,* $W_I : P \times T \longrightarrow \mathbb{N}^* \cup \{\infty\}$*;*
5) *the TPN initial marking,* $M_0 : P \longrightarrow \mathbb{N}$*;*
6) *the earliest firing delays of transitions,* $ED : \ T \longrightarrow \mathbb{Q}^+$*;*
7) *the latest firing delays of transitions,* $LD : \ T \longrightarrow \mathbb{Q}^+ \cup \{\infty\}$.

The classic definition about Petri nets with no time information ranges from items 1 to 5.

Graphically, no regular arc between two nodes (one place and one transition) means that the weight is 0, and a regular arc without a weight label means that the weight is 1. Similarly, no read arc directed from $p \in P$ to $t \in T$ means $W_R(p,t) = 0$, and no inhibitor arc directed from $p \in P$ to $t \in T$ means $W_I(p,t) = \infty$.

A transition t is *enabled* by a marking $M : P \longrightarrow \mathbb{N}$ when: $\forall p \in P, (M(p) \geq W(p,t) \wedge M(p) \geq W_R(p,t)) \wedge \nexists p \in P, M(p) \geq W_I(p,t)$. The set of transitions enabled by M is denoted $En(M)$.

Items 6 and 7 add quantitative time informations: each transition t bears a static interval $[ED(t), LD(t)]$, with $ED(t) \leq LD(t)$, specifying the delay after which t is *fireable* since the instant it becomes enabled or re-enabled.

2.1.2. Semantics

The chosen operational semantics of TPN is the *standard semantics* (as in the references [2, 6]).

The well-known two characterizations of TPN state [5] are interval state and clock state: the second one is used here since it is more general than the first one for building all kinds of abstractions.

TPN semantics is formally defined by means of a timed transition system. Let be a vector $v \in (\mathbb{R}^+)^T$: $v(t_i)$ denotes the *local clock* of transition $t_i \in T$, that is the quantity of elapsed time since transition t_i becomes fireable [2]. $q = (M, v)$ is a state of the timed transition system.

Definition 2. *The* timed transition system $(Q, \{q_0\}, T, \rightarrow)$ *of a marked TPN is defined by:*

- ***1)*** *the initial state is* $q_0 = (M_0, v_0) \in Q$, *with* $v_0 \stackrel{\text{def}}{=} (0)^T$;
- ***2)*** *the set of the states reachable from* q_0 *are* $Q \subseteq (\mathbb{N})^P \times (\mathbb{R}^+)^T$;
- ***3)*** *the alphabet of the discrete transitions is* T;
- ***4)*** *the relation of the timed and instantaneous transitions is* $\rightarrow \subseteq Q \times (T \cup \mathbb{R}^{+*}) \times Q$, *with:*
  - ***a)*** *a timed transition by a delay* $d \in \mathbb{R}^{+*}$ *such as* $(M, v) \stackrel{d}{\rightarrow} (M, v')$ *iff:*

$$\begin{cases} v'(t_i) \stackrel{\text{def}}{=} v(t_i) + d, \ \forall t_i \in T \\ t_i \in En(M) \Rightarrow v'(t_i) \leq LD(t_i) \end{cases}$$

  - ***b)*** *and an instantaneous firing by a transition* $t_i \in En(M)$ *such as* $(M, v) \stackrel{t_i}{\rightarrow} (M', v')$ *iff:*

$$\begin{cases} M' \stackrel{\text{def}}{=} M \backslash {}^\bullet t_i \cup t_i^\bullet \\ ED(t_i) \leq v(t_i) \leq LD(t_i) \\ \forall t_j \in T, \ v'(t_j) \stackrel{\text{def}}{=} \begin{cases} 0 \textit{ if } t_j \in En(M') \\ \quad \wedge (t_j \notin En(M \backslash {}^\bullet t_j) \ \vee t_j = t_i) \\ v(t_j) \textit{ else} \end{cases} \end{cases}$$

In Item 4.b of the definition 2, the notation ${}^\bullet t_i$ (resp. $t_i^\bullet$) represents the multiset [3] of the input (resp. output) places for transition t_i, by the relation W of the regular arcs.

In a state q, any enabled transition is fireable (item 4.b) if its clock value is within the static interval. Thus, an enabled transition t_i must be fired with no more delay if the clock reaches $LD(t_i)$ (item 4.a); otherwise each enabled transition t_i may be delayed by d, as long as any local clock is not increased beyong $LD(t_i)$.

After a firing of some transition t_i, the clock value of each transition in $En(M')$ is updated (item 4.b): it is reset for a *newly* enabled transition $t_j \notin En(M \backslash {}^\bullet t_j) \vee t_j = t_i$, and it is not changed for a *persistent* enabled transition $t_j \in En(M \backslash {}^\bullet t_j)$.

Notice that the clock value of disabled transitions in a state does not mind.

2.1.3. State space abstractions

When time progression is continuous (or dense), the representation of the state-space by a timed transition system is generally infinite. For a bounded TPN (i.e. a marking of a place is always finite), different finite abstractions were proposed [3, 5] by hiding

2. If transition t_i is not enabled, $v(t_i)$ does not mind: this value will not be of no use, to decide equality between two states for instance.

3. A multiset on a set X is a function $Y : X \rightarrow \mathbb{N}$; an equivalent notation is $Y \in (\mathbb{N})^X$.

state reachable by timed transition, according to properties to preserve (especially LTL and CTL). Several states sharing the same marking are agglomerated into a *state class* (a marking with a set of binary time constraints called a *domain*), basing on various conditions of state equivalence. The resulting state-space abstraction is called *state class graph* (SCG): notably, LSCG (Linear SCG) preserves LTL properties, and ASCG (Atomic SCG) preserves CTL properties.

2.2. Grafcet

A GRAFCET chart [9] is a graphical representation to model the behaviour of an automation control part. A GRAFCET chart is made up of two components:

– the *structure* comprises the steps, the transitions and the directed links. The structure describes the possible evolutions between the situations, a *situation* being the set of active steps at a given time;

– the *interpretation* is allowed by the literal variables (inputs, outputs, delays, internal variables, ...) related to the structural elements (steps and transitions). It is done through the transition conditions (containing: inputs, rising/falling edges of boolean inputs, delays, ...) and the actions (continuous or stored actions attached to steps) updating output or internal variables.

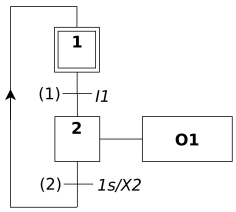

Figure 1. *An example of grafcet.*

The simple example of grafcet Fig. 1 has two steps (square blocks numbered 1 and 2, Step 1 with double square being an initial step), two transitions (1) and (2) respectively with conditions I_1 (an input variable) and $1s/X_2$ (a timed variable on Step 2), and a continuous action block which sets the output O_1 when Step 2 is active. The four directed links connect a step to a transition, or a transition to a step.

Some syntax restrictions are applied to the grafcets for the translation into TPN previously proposed: no hierarchical concepts of forcing and enclosure, no source transition, predicates concern only positive counters (as internal variables), timed variables of the simple form $T_1/X_n/T_2$ or T_1/X_n.

A grafcet transition is *enabled* when each of its input steps is active, and is *fireable* [4] when it is enabled with a true transition condition: the firing provokes simultaneously the activation of all the output steps, and the deactivation of all the input steps (which are not reactivated by the same firing stage). Several simultaneously fireable transitions are simultaneously fired, in the so-called same *firing stage*: this synchronous property is to oppose to the asynchronous firings with Petri nets.

4. The concept of fireability with grafcets is thereby to be differentiated from the one of TPNs.

A situation is *stable* when no transition is fireable. Only external events may cause an evolution, meaning iterated firing stages, from a stable situation. An external [5] event is either an *input change*, or a timed variable change (after some corresponding delay) called *timed event* (or *delay event*).

The algorithm describing the succession of the execution phases of a grafcet is recalled in the work [12]. Likewise, some ambiguous aspects of the standard [9] requiring clarifications and how to carry out stored and continuous actions were presented in this reference.

2.3. About synchronous assumption

Some assumptions about synchronous semantics found the translation principle and are reminded here to prelude the summary about the translation rules.

Every external event may start an evolution, a succession of transition firing stages. An evolution is achieved instantaneously to reach a stable situation, before any next external event which may occur. Thus, the control part of the system which fulfils an evolution, is supposed to be infinitely faster than the rhythm of the environment producing external events. Consequently, the elementary time elapse of the control part is inconsiderable compared to the one of the external clock governing the environment rhythm.

Another consequence of this assumption is that two external events may not occur in the same instant: the reaction of the control part is always completed between two consecutive occurrences of external events. In the specific case when two timed events are synchronous, it is supposed that the control handles them in a total order. It should be noticed that this assumption does not call into question the possibility of parallel execution of several grafcet branches derived from a selection divergence, if the transition conditions allow it.

3. Translation of grafcet into TPN

3.1. The principle

With Petri nets, firings must always be done in a total order: so, the transitions of a firing stage are totally interleaved in the same instant. To allow the simulation of consecutive synchronous firing stages without interference between them, an infinitesimal delay (ε) is introduced, without calling into question the infinitely faster character of the control part [12].

Concretely, two consecutive synchronous firing stages of grafcet transitions are separated by a delay $2\varepsilon_0$, and the updating of variables (step variables and variables updated by stored actions) occurred ε_0 delay after each instant of firing stage of grafcet transitions, and so occurred ε_0 delay before the next firing stage of grafcet transitions in the same evolution phase.

When no more firing stage is possible (meaning the end of an evolution phase), the stable situation is reached a delay $3\varepsilon_0$ after the last firing stage (that is a delay $2\varepsilon_0$ after the last updating): only one occurrence of an external event allows returning to a new evolution phase, possibly empty of firing stage.

To help an overall understanding of the translation rules reminded in the sequel, a Petri net at Fig. 2 depicts how a grafcet dynamic is interpreted. For the transitions with

5. *external* to the processor of the control part of the system.

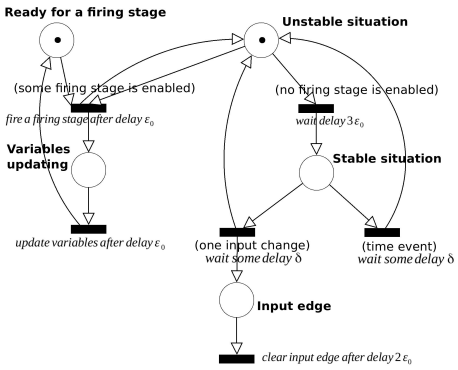

Figure 2. *A Petri net simulating a grafcet behaviour.*

structural conflict in Fig. 2, the choice condition is indicated into brackets. An italic text shows the action of a transition to perform within some delay. Notice that an input rising or falling edge state is just available a delay $2\varepsilon_0$, thereby only for the conditions of the first firing stage of the next evolution phase.

Let make clear the general concept of *stage* after the translation into TPN. A TPN *firing stage* is a set of firings performed in the same instant which simulates a grafcet firing stage. More generally, a stage may also correspond to:

– the set of TPN firings done as variables updating (related to step states or stored actions) during an evolution phase: it is called *update stage*,

– or the synchronous firings done in the instant of coming into a stable situation, to set outputs for continuous actions: it is called *into-stability stage*,

– or in the instant of leaving a stable situation, to set an external event occurrence and to reset outputs for continuous actions: it is called *out-stability stage*.

Thus, a grafcet execution is a cyclic succession of an occurrence of an external event followed by an evolution phase. This cyclic execution is broken down into the four kinds of stages. At the system start-up, a early evolution phase is computed: a first *update stage* initializes diverse variables, which may trigger a *firing stage*, then another update stage, and so on, until reaching a stable situation. Entering a stable situation allows setting the outputs for continuous actions via the *into-stability stage*. At the occurrence of an external event, the *out-stability stage* updates related variables (changing an input or a timed variable) before beginning a new evolution phase: a sub-cycle of a firing stage followed by the subsequent update stage, and so on.

The *head firing* of a stage is the first firing to begin this stage.

These new denominations are used in the sequel to manage the state-space abstractions.

3.2. Translation rules

The application of translation rules follows a suitable order to complete the resulting ε-TPN of a grafcet: the grafcet elements are converted into connected TPN modules composing the ε-TPN. An additional and first module called *phase sequencer* (Fig. 3) allows alternation between an occurrence of an external event (the place $Stable$ is marked to denote a stable situation before producing an external event) and the subsequent reaction phase (marking of the place $Evolution$ denotes the transient evolution phase following

the external event occurrence), without any interference: especially, no input as external event may not be modified during a transient evolution. The place $Occurrence$ is marked every time a grafcet transition model is fired or when a timed event occurs, allowing to stay in (or to restart) an evolution phase.

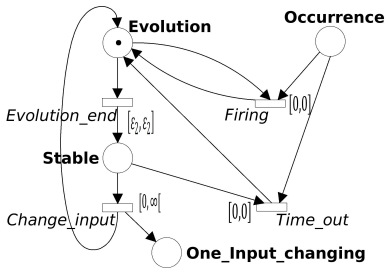

Figure 3. *Phase sequencer.*

After adding this first module, creation of the TPN modules may follow this suitable order: steps, inputs, timed variables, outputs, counter variables, continuous actions, stored actions, grafcet transitions.

The static time interval beared by every transition t of a module get the form of a simple delay $\delta(t)$: $\delta(t) = ED(t) = LD(t)$; except for the transiion named $Change\_input$ in Fig. 3: the firing of this transition with interval $[0, \infty[$ allows the occurrence of an input change at any time.

The given (non exhaustive) elements of translation are respectively, at Fig. 4:

– (a): a step i with state variables X_i and $\overline{X_i}$,

– (b): an input I_x with rising edge $\uparrow I_x$ and falling edge $\downarrow I_x$,

– (c): a timed variable T_j/X_i such as the firing $T\_j\_X\_i\_to\_true$ expresses an external timed event,

– (d): an example of two continuous actions, with the condition X_1 in step 3,

– (e): an example of two transitions, with one shared input step and the condition $I_1 + I_2$ on transition (2).

For each figure, elements in gray are already contained in a previously generated module.

Here, the translation rules are not modified, except slighty for the extra module of the phase sequencer (Fig. 3): $ED(Change\_input)$ which previously has ε_0 value is now replaced by 0. The former value was a trick to make firings related to continuous actions to occur ε_0 delay before a next input change, which is useless now since the transition $Change\_input$ has lesser priority (see Section 4). Moreover, in the previous work [12], it was possible in a stable situation for an input event to occur just before a timed event in the same instant to start a subsequent evolution phase, whereas the converse (i.e. a timed event followed in the same instant by an input event) was not possible. This contradiction is emended here, since the respect of the synchronous assumptions (cf. subsection 2.3) prevents multiple occurrences of external events between two evolution phases.

3.3. An example of translation

As an illustration, the translation into TPN of the example of grafcet at Fig. 1 is provided Fig. 5(b). Two modules are simplified to get the resulting TPN: the step modules

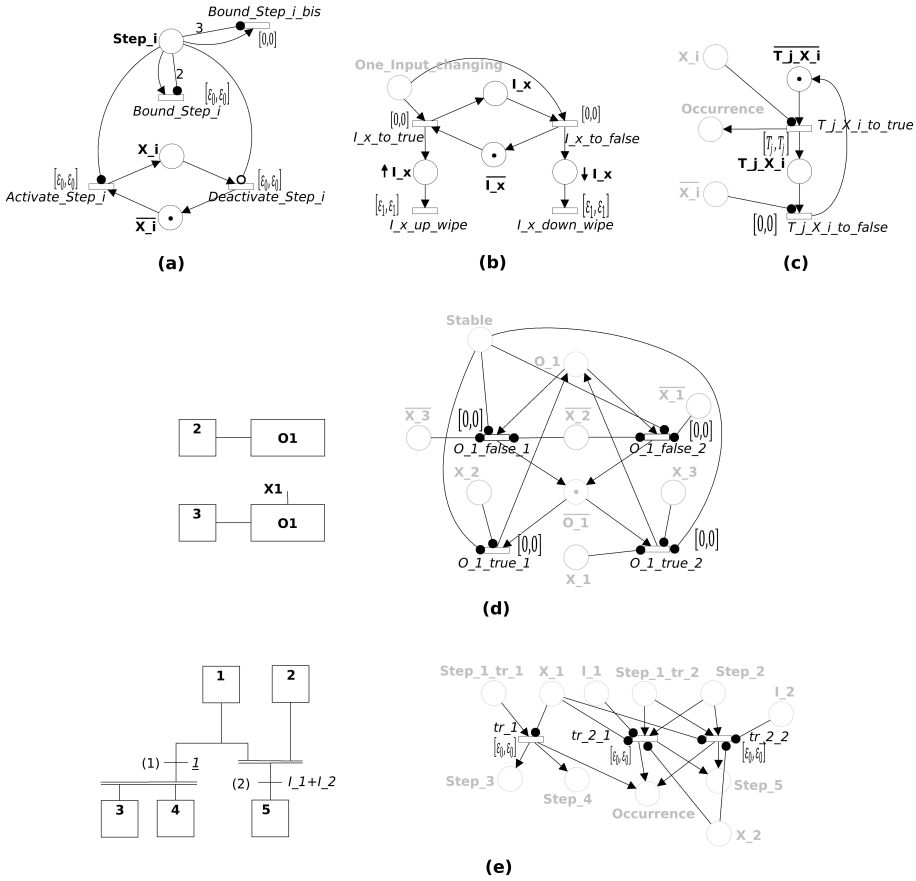

Figure 4. *Examples of translated elements of grafcet.*

because the grafcet model is safe, and the input module for I_1 since the rising and falling edges are useless.

Definition 3. *T_{input} and T_G denote respectively the transitions modelling input changes and grafcet transitions.*

For the given example, $T_{input} = \{I\_1\_to\_true, I\_1\_to\_false\}$ and $T_G = \{tr\_1, tr\_2\}$. Incidentally, $T_T = \{1s\_X\_2\_to\_true\}$.

3.4. Enhancement on stored action modeling

Here, the goal is to modify the modelling of the ordering of a stored action on a boolean variable (internal or output). Indeed, the actual model given at Fig. 6(c) allows an unpleasant behaviour: when the grafcet contains undesirably contradictory orders on set and reset the boolean variable B_i in a same situation (one token marking of the places Set_B_i and $Reset\_B\_i$ at Fig. (b) corresponds to the orders in Fig. (a)), the TPN translation at Fig. (c) allows to fire consecutively the transitions $B\_i\_\_to\_true$ and $B\_i\_\_to\_false$ in an infinite loop (in zero time), meaning that an evolution phase may last indefinitely.

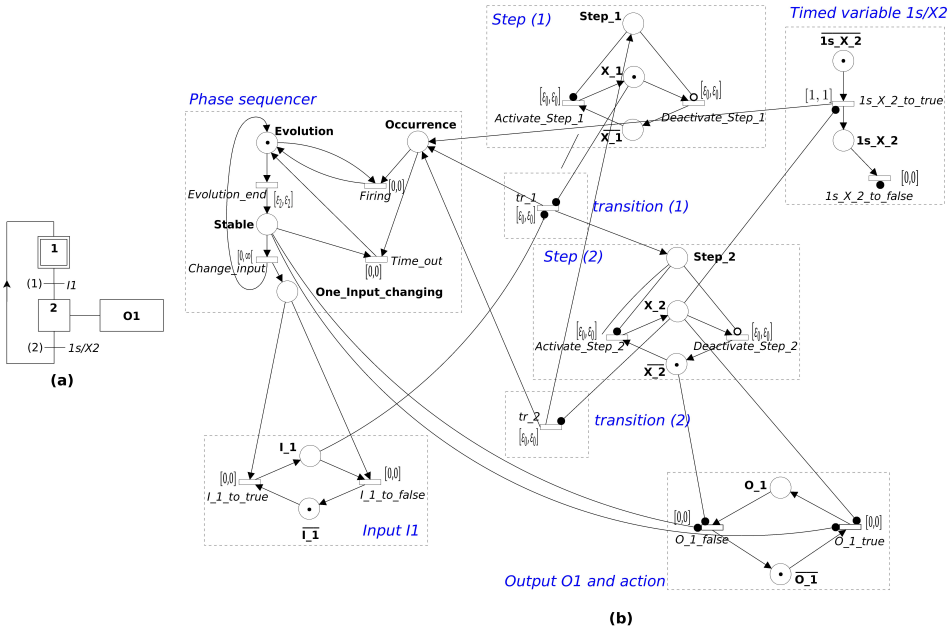

Figure 5. *The example of grafcet at Fig. 1 (a) and the translation into ε-TPN (b).*

To avoid such a behaviour involving specifically a non-finite update stage, two inhibitor arcs are added at Fig. 6(d) representing the new modelling: a set (resp. reset) order prevents now resetting (resp. setting) B_i in a current update stage. Of course, the model checking should spot the problem for afterward corrective action in the grafcet model.

4. Formal definitions on ε-TPN

In this section is formally defined the TPN extension, called ε-TPN, used as the target model of the grafcet to translate. Moreover, the specificity of the grafcet semantics and how the ε-TPN modules translate the entities of a grafcet justify some stated features: the subsequent algorithms about state-space abstraction are based on these features.

4.1. Syntax

Let ε_0 be an infinitesimal constant delay comparable to 0^+. For $n \in \mathbb{N}$, $\varepsilon_n \stackrel{\text{def}}{=} \varepsilon_0 \times (n+1)$, and $\mathcal{E} \stackrel{\text{def}}{=} \{\varepsilon_n \mid n \in \mathbb{N}\}$. For $\varepsilon_n \in \mathcal{E}$ and any $d \in \mathbb{R}^{+*}$, it is assumed that: $0 < \varepsilon_n < d$ and $d \pm \varepsilon_n \approx d$. And $\mathcal{E}_0 \stackrel{\text{def}}{=} \mathcal{E} \cup \{0\}$ by extension.

Definition 4. *An ε-TPN is a TPN with specific static firing intervals for transitions:*

1) *the earliest firing delays of transitions, $ED : T \longrightarrow \mathcal{E} \cup \mathbb{Q}^+$;*

2) *the latest firing delays of transitions, $LD : T \longrightarrow \mathcal{E} \cup \mathbb{Q}^+ \cup \{\infty\}$;*

3) *the set of transitions is made of three subsets, $T = T_{\mathcal{E}_0} \cup T_T \cup T_\infty$ such as:*

- ***a)*** *$ED(t) = LD(t) \in \mathcal{E}_0$ when $t \in T_{\mathcal{E}_0}$;*
- ***b)*** *$ED(t) = LD(t) \in \mathbb{Q}^{+*}$ when $t \in T_T$;*

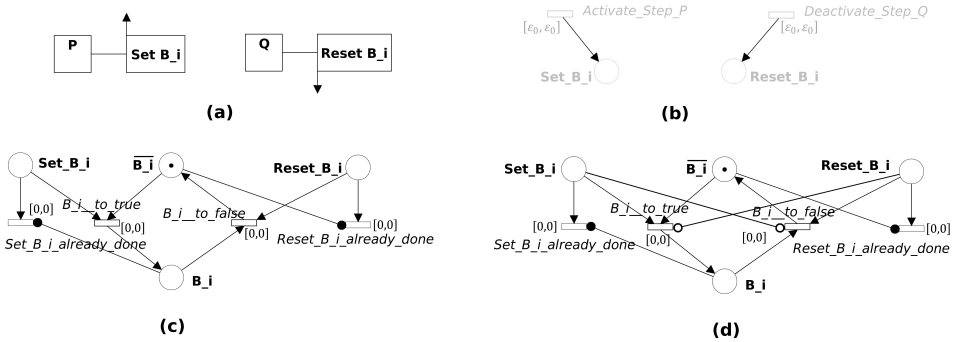

Figure 6. *Stored action.*

c) $ED(t) = 0$ *and* $LD(t) = \infty$ *when* $t \in T_\infty$.

Thus, definition 4 extends the classic definition 1 with use of the infinitesimal delays (here, $\mathcal{E} \stackrel{\text{def}}{=} \{\varepsilon_0, \varepsilon_1, \varepsilon_2\}$) in the transition bounds, but restricts the form of the static firing intervals for transitions which mostly allow only fixed delays.

In the sequel, $T_\infty = \{Change\_input\}$ is a single transition denoted by t_∞: the variable delay experienced before such a firing represents the time spent in a stable state before the occurrence of the next external event.

Transitions in $T_\mathcal{E}$ model synchronous firings done in an evolution phase. External events to the control part depend on the transitions $\{t_\infty\}$ (an input event occurs in the same instant of this firing) and T_T (delay events for the timed variables): two transitions of these kinds may never occur simultaneously in the same instant, as imposed by the operational semantics. Thus, only one firing in the set $\{t_\infty\} \cup T_T$ should trigger a reaction, that is a sequence of firings in $T_{\mathcal{E}_0}$. The transitions with zero delay value (i.e. with static interval $[0, 0]$) may also be used to produce an external event, in order to update some variables instantaneously.

Subsequently, the fixed delay of a transition $t \in T_{\mathcal{E}_0} \cup T_T$ is denoted $\delta(t)$.

4.2. Operational semantics

If transitions are enabled in both the sets $T_{\mathcal{E}_0}$ and $T_\infty \cup T_T$, then transitions in $T_{\mathcal{E}_0}$ always have priority to be fired, contrary to the previous work [12] which considers no such priority. When a transition in $T_\infty \cup T_T$ is really fireable, dense time elapse is allowed in a *stable* state, by a non-infinitesimal delay. After such an external event, a reaction phase is entered where only firings in $T_{\mathcal{E}_0}$ are possible, before to reach a next stable state.

Compared to previous definition 2, a vector v gets its values in $\mathcal{E} \cup \mathbb{R}^+$: $v \in (\mathcal{E} \cup \mathbb{R}^+)^T$.

Definition 5. $(Q, \{q_0\}, T, \to)$ *for a marked* ε*-TPN is defined by:*

1) *the initial state is* $q_0 = (M_0, v_0) \in Q$*, with* $v_0 \stackrel{\text{def}}{=} (0)^T$*;*

2) *the set of states reachable from* q_0 *are* $Q \subseteq (\mathbb{N})^P \times (\mathcal{E} \cup \mathbb{R}^+)^T$*;*

3) *the alphabet of the discrete transitions is* T*;*

4) *the relation of the timed and instantaneous transitions is* $\to \subseteq Q \times (T \cup \mathcal{E} \cup \mathbb{R}^{+*}) \times Q$*, with:*

a) *a timed transition by a delay* $d \in \mathcal{E} \cup \mathbb{R}^+$ *such as* $(M, v) \stackrel{d}{\to} (M, v')$*:*

i) *if* $\exists d \in \mathcal{E}$ *(infinitesimal time elapse), then:*

$$\begin{cases} \exists t \in En(M) \cap T_{\mathcal{E}} \\ \forall t_j \in T, \begin{cases} \textit{if } t_j \in T_{\mathcal{E}_0} \\ \quad \textit{then } v'(t_j) \stackrel{\text{def}}{=} v(t_j) + d, \\ \quad \textit{else } v'(t_j) \stackrel{\text{def}}{=} v(t_j) \\ t_j \in En(M) \Rightarrow v'(t_j) \leq LD(t_j) \end{cases} \end{cases}$$

ii) *if* $\exists d \in \mathbb{R}^{+*}$ *(non-infinitesimal time elapse), then:*

$$\begin{cases} \nexists t \in En(M) \cap T_{\mathcal{E}_0} \\ v' \stackrel{\text{def}}{=} v + d \\ \forall t_j \in En(M) \Rightarrow v'(t_j) \leq LD(t_j) \end{cases}$$

b) *and an instantaneous firing by a transition* $t_i \in En(M)$ *such as* $(M, v) \stackrel{t_i}{\rightarrow} (M', v')$ *iff:*

$$\begin{cases} t_i \in T_{\mathcal{E}_0} \vee (\nexists t_j \in En(M) \cap T_{\mathcal{E}_0}) \\ \quad \wedge (t_i \in T_T \cup T_\infty) \\ M' \stackrel{\text{def}}{=} M \backslash {}^\bullet t_i \cup t_i^\bullet \\ ED(t_i) \leq v(t_i) \leq LD(t_i) \\ \forall t_j \in T,\ v'(t_j) \stackrel{\text{def}}{=} \begin{cases} 0 \textit{ if } t_j \in En(M') \\ \quad \wedge\ (t_j \notin En(M \backslash {}^\bullet t_j) \vee t_j = t_i) \\ v(t_j) \textit{ else} \end{cases} \end{cases}$$

Thus, when the set $En(M) \cap T_{\mathcal{E}_0}$ is not empty in a given state q, a transition t in this set is fired instantaneously if the clock value $v(t) = \delta(t)$, or the least infinitesimal elapse is observed to make one of them fireable. Meanwhile, clock values do not change for enabled transitions in $T_T \cup T_\infty$. Otherwise, only transitions in $T_T \cup T_\infty$ are enabled, and the common semantics for TPN is applied (definition 2). After a firing, the clocks value of the transitions $En(M')$ are updated in accordance with the standard semantics.

4.3. Adequate features on the ε-TPN

Here is highlighted some features resulting from the translated grafcet into ε-TPN, features on which the subsequent developments rely on, the state-space abstractions especially.

The equivalence between a GRAFCET chart and the TPN resulting from the translation was shown [12]. In particular, an induced first feature is the *boundness* of the TPN, which determines the construction of a finite state-space. Boundness implies that each modelled variable ranges in a finite domain: only an unbounded counter in a grafcet may call into question this feature.

4.3.1. Deadlock-freeness

This feature results from the phase sequencer (Fig. 3): evolution phase and stable phase alternate indefinitely, even if an evolution phase between two external events may be empty of grafcet transition firing (and so empty of internal variables updating): the transition $Evolution\_End$ is just fired.

4.3.2. A stage is finite

This feature results from the structure of the generated finite and bounded TPN. Indeed, only a structural loop of transitions with zero static delay may cause an infinite

stage. That is to be check locally on modules of translation, because it is not difficult to apprehend graphically that firings allowed by interconnections between two modules always imply a non zero static delay transition. Modules with loop by zero static delay transitions are:

– the phase sequencer, with the transition $Firing$ (Fig. 3): only firings from a grafcet transition or a transition in T_T (in a timed variable module), which are in finite number, allow this firing;

– an input module (Fig. 4(b)): one token put in the place $One\_Input\_Changing$ by the transition t_∞ causes only one transition firing in all input modules (T_{input});

– a continuous action module (Fig. 4(d)): a firing depends on the related step state (X_i or $\overline{X_i}$), so only one is possible;

– a stored action module on a boolean variable (the new proposal at Fig. 6(d)): the subsection 3.4 justifies the finiteness of a such execution.

4.3.3. A non-finite evolution phase is always detectable

A *total instability* is when an evolution phase lasts indefinitely. It compromises the synchronous assumption since this evolution will not end before some next external event. By saving and comparing states reached by each finite firing or update stage constituting a current evolution, a loop is detectable.

4.3.4. Firings which are relevant to an evolution phase may not be interfered with the occurrence of an external event

Initially, a first evolution phase is executed, and transitions $T_T \cup T_{input}$ are not fireable. Then, every time a stable state is reached, an external event $t_e \in T_T \cup \{t_\infty\}$ is produced to begin a new evolution phase. No interference with the first firing stage (after a delay ε_0) of the evolution phase is possible because the remaining firings due to the produced external event are done in the same instant of the related out-stability stage: the transition $Time\_out$ if $t_e \in T_T$, or one of the transition T_{input} if $t_e = t_\infty$. In short, priority avoids possible interference.

4.3.5. In a stable state, besides transitions $T_T \cup \{t_\infty\}$, only transitions with static interval $[0, 0]$ may be enabled

Before producing an external event in a stable state, one token in the place $Stable$ (caused by the transition $Evolution\_end$, when leaving the evolution phase) may allow firings of zero static delay transitions of continuous action modules [6]. Indeed, no more grafcet or update transition enabled in an evolution phase remains fireable before firing $Evolution\_end$.

Besides, a firing in T_T makes the zero static delay transition $Time\_out$ to be enabled, transition of which firing causes leaving the stable state; whereas the firing of t_∞ causes immediate leaving of the stable state.

4.3.6. When a transition in $T_T \cup \{t_\infty\}$ is fired, then only transitions with zero static delay are fireable in the same out-stability stage

It is contained in the development of the features 4.3.4 and 4.3.5.

6. Notice that in an evolution phase, a firing in a continuous action module is impossible because it depends on the marking of the place $Stable$.

4.3.7. When the transition t_∞ is fired, then one transition in T_{input} for each input defined in the grafcet is fireable in the same out-stability stage

It is contained in the development of the features 4.3.4 and 4.3.2.

4.3.8. When the transition t_∞ is fireable, its clock value does not change by a concurrent firing in T_T

t_∞ is fireable after zero-delay firings for stored actions (during an into-stability stage); then, it remains *a priori* persistent when a transition in T_T is fired at first (to head an out-stability stage). But a firing in T_T causes the enabling of $Time\_out$ having priority on t_∞ and fired without delay. Since this firing is in conflict with t_∞ (by sharing the place $Stable$), t_∞ is disabled. Not doing a firing in T_T means t_∞ will necessarily be fired: an observed delay is only due to waiting this firing.

5. State space abstractions

5.1. Specificity of ε-TPN abstractions

Each of the four kinds of stages, presented in subsection 3.1, is made of a finite number of firings (sequential and/or concurrent) done in the same instant (according to feature 4.3.2 in the previous subsection), and the possible interleavings of the synchronous firings should lead up to the same state. So, to manage the state explosion problem, only one interleaving should be dealt with, while insuring (if necessary) that the stage always ends up in a unique and coherent state, by an unexpensive way of checking it: subsection 5.5 suggests a way to check a coherent execution of a stage, so-called the *consistency check*.

The first level of abstraction, called *partial-order state-space*, consists in computing all the reachable state classes by the computed stages, from the initial class C_0. In addition to this first level of abstraction, two subsequent levels more relevant for a grafcet point of view are considered, by abstracting the informations too specific to the target ε-TPN model:

– the all situations state-space, containing all the stable and unstable situations of the grafcet: every firing stage followed by an update stage are merged into one stage,

– and the stable situations state-space, only containing the stable situations of the grafcet: all consecutive stages between two stable states are merged into one.

In practice, informations in the classes and in the merged stages of these two kinds of abstraction should be filtered and simplified (as far as renaming places and transitions modelling the grafcet) to only keep relevant informations about the original grafcet. However, these conveniences which purpose is to improve the transparence of the state-space towards the TPN formalism, are taken into account only partially in the sequel: namely for the transitions of these two high level state-spaces.

The boundness of the ε-TPN may be called into question only by an unbounded counter variable modelling. So, the test of the state-space finiteness in this eventuality should be easily integrated in the algorithms, in the classical manner with TPNs [2]: here, a state-space abstraction is finite iff the ε-TPN is bounded.

5.2. Composite classes

Before presenting the three levels of abstractions, some accommodations are necessary on the usual ways to compute the state class graph (SCG), because of the specificity of a ε-TPN.

So, an extended definition of class is justified by integration of discrete infinitesimal time delay ε_0 for $T_{\mathcal{E}}$ transitions. In fact, for a domain interval such as $[0, \varepsilon_0]$, time could only take two values, 0 or ε_0, and intermediate values are nonsensical since time is no more dense. Here, we choose to keep the classic definition of a class domain by considering 0 instead of ε_n for the enabled $T_{\mathcal{E}}$ transitions. But, another time informations about discrete clock values of the enabled transitions in $T_{\mathcal{E}_0}$ are needed to know which transitions are really fireable among ones having priority.

A *composite* state class C is a tuple of a marking M, a clock domain D (a conjunction of dense time binary constraints between the clock values of all enabled transitions) and a set of discrete clock values V for the enabled transitions in $T_{\mathcal{E}_0}$. The clock variable of a transition t_i appearing in a global domain D (resp. in the set V) is denoted by τ_i (resp. ν_i). The initial class is $C_0 = (M_0, D_0, V_0)$, such as $D_0 = \bigwedge_{t_i \in En(M_0)} \tau_i = 0$ and $V_0 = \{\nu_i = 0 \mid t_i \in En(M_0) \cap T_{\mathcal{E}_0}\}$.

For a current class C, if $En(M) \cap T_{\mathcal{E}_0}$ is empty (the set V is empty), the fireability check of each $t_i \in En(M)$ and the next class C' reached by fireable t_i are computed as usual [5]. Else (i.e. $En(M) \cap T_{\mathcal{E}_0} \neq \emptyset$), a transition t_f having priority must be fired, in the set $\{t_f \mid \forall (t_f, t_i) \in (En(M) \cap T_{\mathcal{E}_0})^2, \delta(t_f) - \nu_f \leq \delta(t_i) - \nu_i\}$; the next class domain D' is still computed as usual (by considering 0 bounds for $T_{\mathcal{E}}$ transitions) and the set V' as follows: $\forall t_i \in En(M') \cap T_{\mathcal{E}_0}$, $\nu'_i = 0$ if newly enabled, $\nu'_i = \nu_i + (\delta(t_f) - \nu_f)$ else. Indeed, a discrete delay $(\delta(t_f) - \nu_f)$ is observed between the classes C and C'.

An ε-TPN gets an unbounded static interval, the only one of the transition t_∞. Thus, the risk is an infinite SCG exits, justifying the relaxation operation when using clock state based abstraction [3], which is necessary for ASCG. Fortunately, transition t_∞ may not induce an infinite number of domains: feature 4.3.8 justifies no need of relaxation, thanks to mutual exclusion firing between t_∞ and T_T transitions.

As illustration, the SCG (without partial order execution of the stages) of the example Fig. 5(b) is made of 41 classes and 50 transitions.

5.3. Abstraction of level 0: Partial-order state-space

To get the level 0 SCG (L0-SCG), a transition between two classes C and C' is a multiset T_s of the firings in a stage ($C \stackrel{T_s}{\rightarrow} C'$): the intermediate states during a stage are not saved.

The initial class C_0 corresponds to the initial state of the grafcet (initial situation in an evolution phase, with the setting of the initial value of the diverse variables). So, the first stage to fire is a firing stage if the initial situation is not stable; otherwise, the transition $Evolution\_end$ is fired to trigger the first into-stability stage. The head firing of a stage is always the transition $Evolution\_end$ (denoted t_{ε_2} in the sequel) for an into-stability stage, and is a transition t_∞ or a delay transition in T_T (modeling external events) for an out-stability stage.

For a partial-order state-space of an unambiguous grafcet, it should be noticed that only classes from which an external event is fireable may get multiple outgoing stage transitions in L0-SCG. Every firing interleaving of a stage should lead up to the same class, except when the head firing is a transition t_∞ for an out-stability stage: the real input

event is produced by one change among the possibly multiple inputs in T_{input} (feature 4.3.7). Therefore, a class from which a transition t_∞ is enabled will get as many outgoing transitions as the number of existing inputs.

```
1  Input: marked ε-TPN;
2  Output: sets C_{L0} and T_{L0} for L0-SCG;
3  C_{L0} := {C_0}; T_{L0} := {};
4  Stack C_0;
5  while the stack is not empty do
6  |  Unstack C = (M, D, V);
7  |  if ∃t ∈ En(M) ∩ T_{ε0} then
8  |  |  FromEvolution();
9  |  |  if C' ∉ C_{L0} then Stack C';
10 |  |  AddSCG(C, T_s, C');
11 |  else
12 |  |  FromStable() ;
13 |  end
14 end
```

Algorithm 1: Construction of the L0-SCG

Algorithm 1 describes the construction of the L0-SCG. It is globally structured like the classical algorithm computing a PN state graph, but with lines 7-13 specialized to stages computing (and saving only classes reached by the stages) as macro-transitions instead of simple transition firings. Stages are computed by calls to procedures FromEvolution at line 8 and FromStable at line 12. FromEvolution (resp. FromStable) implicitely deals with firing stages, update stages and in-stability stages (resp. out-stability stages). None of these procedures is provided with test of execution finiteness (or loop detection) thanks to the feature 4.3.2. Moreover, because of the deadlock-freeness (feature 4.3.1), a head firing t is always found to begin a stage from a new class C to treat from the stack: either a transition having priority at line 7, or an external event at line 11.

```
1  Find any fireable transition t' from C;
2  Compute the successor class C'; T_s := {{t'}};
3  while ∃t'' ∈ En(M') ∩ T_{ε0}, with t'' and t' fireable at the same time do
4  |  Compute C'' from C'; T_s := T_s ∪ {{t''}}; C' := C'';
5  end
```

Procedure FromEvolution

If the class reached by the stage computed by the procedure FromEvolution at line 8 of Algorithm 1 is a new one, this class is stacked at line 9 to handle it later, and next at line 10 is saved the new transition caused by this stage with the call to the procedure AddSCG.

```
1  Input: C, T_s, C';
2  if C' ∉ C_{L0} then
3  |  C_{L0} := C_{L0} ∪ {C'}; T_{L0} := T_{L0} ∪ {(C, T_s, C')};
4  end
5  T_{L0} := T_{L0} ∪ {(C, T_s, C')};
```

Procedure AddSCG

```
1  foreach t' ∈ En(M) fireable from C do
2      Compute C' from C; T_s := {{t'}};
3      if t' = t_∞ then
4          foreach transition t'' ∈ En(M') ∩ T_input do
5              Compute C'' from C'; T_s := T_s ∪ {{t''}};
6              if C'' ∉ C_{L_0} then Stack C'';
7              AddSCG(C, T_s, C'');
8          end
9      else
10         while ∃t'' ∈ En(M') ∩ T_{ε_0} such as δ(t'') = 0 do
11             Compute C'' from C'; T_s := T_s ∪ {{t''}}; C' := C'';
12         end
13         if C' ∉ C_{L_0} then Stack C';
14         AddSCG(C, T_s, C');
15     end
16 end
```

Procedure FromStable

Lines 7-10 of Algorithm 1 deal with any stage, except the out-stability stage which is managed at line 12 with the call to the procedure FromStable: the **if** block of the procedure (lines 3-8) deals with firing t_∞ followed by a firing in the set T_{input} in the same instant (feature 4.3.7); the **else** block (lines 9-15) deals with a timed event firing in T_T followed by firings of transition with zero static delay (feature 4.3.6), in fact a singleton set made of the transition $Time\_out$ (feature 4.3.5) to return to an evolution phase.

The usage of a stack by Algorithm 1 to save classes is pertinently justified when interleavings are necessary for possible multiple external events in a stable state (lines 6 and 13 of the macro FromStable). The AddSCG procedure call at lines 7 and 14 of the macro FromStable allows to save relevant classes and the related transitions as stages.

The consistency check of a stage evoked in subsection 5.1 may be useful, especially for an update stage: the procedure is presented in the subsection 5.5.

The procedures FromEvolution, FromStable, and AddSCG are reused for the following levels of abstraction in the next subsection. Notice that it is supposed SCG data in Algorithm 1 are shared with the procedures, as the current class informations (C, C', T_s, ...) for the macros FromStable and FromEvolution.

Besides, it may be observed that the stack used for Algorithm 1 contains as well stable states as unstable (i.e. evolution) states. And the possibility of a *total instability* of the grafcet model evoked in subsection 4.3, does not mind yet here; this is taken into account in the sequel.

As illustration, the L0-SCG of the example Fig. 5(b) is made of 18 classes and 21 transitions, detailed in Table 1.

5.4. Abstractions of levels 1 and 2: grafcet situations state-space

Algorithm 2 describes the constructions of the L1-SCG (level 1 SCG, with the sets $\mathcal{C}_{L_1}$ and $\mathcal{T}_{L_1}$) and L2-SCG (level 2 SCG, with the sets $\mathcal{C}_{L_2}$ and $\mathcal{T}_{L_2}$), which are worth doing as grafcet state-space. These constructions are more largely based on the knowledge of the possible succession order of the different stages induced by a grafcet semantics:

– a firing stage is always followed by an update stage: these consecutive stages are merged in practice for a grafcet, and the intermediate state between them does not need

Table 1. *L0-SCG transitions of the example Fig. 5(b).*

Transitions	T_s	Transitions	T_s	Transitions	T_s
$C_0 \overset{T_s}{\rightarrow} C_1$	t_{ε_2}	$C_6 \overset{T_s}{\rightarrow} C_7$	t_{ε_2}	$C_{11} \overset{T_s}{\rightarrow} C_{12}$	$Activate\_Step\_1$, $Deactivate\_Step\_2$, $1s\_X\_2\_false$
$C_1 \overset{T_s}{\rightarrow} C_2$	$t_\infty, I\_1\_to\_true$	$C_7 \overset{T_s}{\rightarrow} C_8$	$t_\infty, I\_1\_to\_true$	$C_{12} \overset{T_s}{\rightarrow} C_{13}$	t_{ε_2}
$C_2 \overset{T_s}{\rightarrow} C_3$	$tr\_1, Firing$	$C_7 \overset{T_s}{\rightarrow} C_{15}$	$1s\_X\_2\_true$, $Time\_out$	$C_{13} \overset{T_s}{\rightarrow} C_{14}$	$Deactivate\_Step\_1$, $Activate\_Step\_2$
$C_3 \overset{T_s}{\rightarrow} C_4$	$Deactivate\_Step\_1$, $Activate\_Step\_2$	$C_8 \overset{T_s}{\rightarrow} C_9$	t_{ε_2}	$C_{14} \overset{T_s}{\rightarrow} C_{15}$	t_{ε_2}
$C_4 \overset{T_s}{\rightarrow} C_5$	$t_{\varepsilon_2}, O\_1\_true$	$C_9 \overset{T_s}{\rightarrow} C_6$	$t_\infty, I\_1\_to\_false$	$C_{15} \overset{T_s}{\rightarrow} C_{16}$	$tr\_2, Firing$
$C_5 \overset{T_s}{\rightarrow} C_6$	$t_\infty, I\_1\_to\_false$	$C_9 \overset{T_s}{\rightarrow} C_{10}$	$1s\_X\_2\_true$ $Time\_out$	$C_{16} \overset{T_s}{\rightarrow} C_{17}$	$Activate\_Step\_1$, $Deactivate\_Step\_2$, $1s\_X\_2\_false$
$C_5 \overset{T_s}{\rightarrow} C_{10}$	t_{ε_2}	$C_{10} \overset{T_s}{\rightarrow} C_{11}$	$tr\_2, Firing$	$C_{17} \overset{T_s}{\rightarrow} C_1$	$1s\_X\_2\_true$, $Time\_out$

Input: marked ε-TPN;
Output: sets $\mathcal{C}_{L_1}$ and $\mathcal{T}_{L_1}$ (L1-SCG), sets $\mathcal{C}_{L_2}$ and $\mathcal{T}_{L_2}$ (L2-SCG);
$\mathcal{C}_{L_1} := \{C_0\}$; $\mathcal{C}_{L_2} := \{\}$; $C = C_0$;
$\mathcal{T}_{L_1} := \{\}$; $\mathcal{T}_{L_2} := \{\}$;
repeat
  FromEvolution(); `// Firing Stage`
  $C_* := C$; $C := C'$; $T_{*,s} := T_s \cap T_G$;
  FromEvolution(); `// Update Stage`
  Make the consistency check of the stage T_s;
  if $C' \in \mathcal{C}_{L_1}$ **then exit (NEVER_STABILITY)** ;
  AddSCG1($C_*, T_{*,s}, C'$); $C = C'$;
until t_{ε_2} *is fireable from* C;
FromEvolution(); `// Into-stability Stage`
AddSCG1($C, \{t_{\varepsilon_2}\}, C'$); Stack C' in STACK1;
while *STACK1 is non empty* **do**
  Unstack C_S from STACK1; $C = C_S$;
  FromStable2(); `// Out-stability Stage`
  while *STACK2 is not empty* **do**
    Unstack (T_{ext}, C) from STACK2;
    $Loc\_Classes := \{C\}$;
    repeat
      FromEvolution(); `// Firing Stage`
      $C_* := C$; $C := C'$; $T_{*,s} := T_s \cap T_G$;
      FromEvolution(); `// Update Stage`
      Make the consistency check of the stage T_s;
      if $C' \in Loc\_Classes$ **then exit (NO_STABILITY)** ;
      if $C_* \notin \mathcal{C}_{L_1}$ **then** AddSCG1($C_*, T_{*,s}, C'$) ;
      $Loc\_Classes := Loc\_Classes \cup \{C'\}$; $C := C'$;
    until t_{ε_2} *is fireable from* C;
    FromEvolution(); `// Into-stability Stage`
    if $C \notin \mathcal{C}_{L_1}$ **then** AddSCG1($C, \{t_{\varepsilon_2}\}, C'$) ;
    if $C' \notin \mathcal{C}_{L_2}$ **then** Stack C' in STACK1 ;
    AddSCG2(C_S, T_{ext}, C');
  end
end

Algorithm 2: Construction of the L2-SCG

to be saved (lines 6-8 and 22-24);

– class C_0 starts an evolution phase which eventually ends with a first stable situation (lines 5-12);

– any total instability should be detected (at lines 10 and 26) to abort the SCG computing.

For L1-SCG, the procedure AddSCG1 replaces AddSCG (at lines 11, 27 and 31 in Algorithm 2, and at lines 7 and 14 in procedure FromStable2 replacing FromStable) by rather acting on the sets $\mathcal{C}_{L_1}$ and $\mathcal{T}_{L_1}$ (instead of the sets $\mathcal{C}$ and $\mathcal{T}$ with Algorithm 1). The first *stable state class* (i.e. a state class such as the place $Stable$ is marked, meaning a state of a stable grafcet situation) is saved on STACK1 at line 14: STACK1 contains all the stable state classes to be treated, by computing their successor classes. For each class pulled from STACK1 (line 16), the next classes reached by each possible external event with the related out-stability stage are saved on STACK2 by the macro FromStable2 replacing FromStable, with the same code lines but with some name substitutions. Indeed, lines 6 and 13 for the macro FromStable2 acts now on STACK2 (instead of $Stack$ with Algorithm 1) which contains only classes reached by an out-stability stage, that is the states beginning an evolution phase. Besides, the set $T_{ext} = T_s \cap (T_{input} \cup \{t_\infty\})$ (resp. $T_{ext} = T_s \cap T_T$) replaces T_s in the call AddSCG1 at line 7 (resp. line 14) of the macro FromStable2. Precisely, STACK2 saves a couple (T_{ext}, C) instead of a simple class C at lines 6 and 13 in the macro FromStable2.

Each item of the STACK2 is treated (lines 18-34 of Algorithm 2), allowing thereby to compute the successive classes until reaching the next stable state class, saved in STACK1 if new (line 32). To detect a possible total instability (always detectable according to the feature 4.3.3) which might prevent reaching some next stable state class, the following classes encountered (and only derived from the current stable state class C_S to treat) are stored in the set $Loc\_Classes$: so, if a loop back is done before reaching an expected stable state, that would mean that the instability is never left, and therefore the algorithm is stopped (line 26). The same precaution is taken to reach the first stable state class from C_0 (line 10), by using directly the set $\mathcal{C}_{L_1}$ which is not derived from multiple stable state classes meanwhile. Notice that constituting $Loc\_Classes$ is not subject to explosion since no interleaving is considered before reaching the next stable state class.

L2-SCG is constituted only at the line 33 of Algorithm 2 with the macro AddSCG2: only the external event (T_{ext}) minds between two stable state classes. It should be mentioned that a transition T_s between two classes, as well for L1-SCG as for L2-SCG, is now a simple set to only reflect the grafcet model elements: either T_{ext}, or $\{t_{\varepsilon_2} = Evolution\_end\}$ (from an evolution phase to a stable state, lines 12-14 and 29-31) or $T_s \cap T_G$ (for a firing stage, with the subsequent update stage, at lines 6-8 and lines 22-24).

If only L2-SCG minds as the state-space to generate, then the calls to the macro AddSCG1 should be discarded in Algorithm 2 and in procedure FromStable2.

The consistency check is done for an update stage at lines 9 and 25.

At lines 14 and 31, although the transition $T_s = \{t_{\varepsilon_2}\}$ is specific to ε-TPN, it expresses the leaving of an evolution phase, and it may be useful to detect a deadlock in the source grafcet (i.e. an always empty evolution phase).

As illustration, the L1-SCG of the example Fig. 5(b) is made of 14 classes and 17 transitions: it just saves the classes $\{C_3, C_{11}, C_{13}, C_{16}\}$ in the L0-SCG. The L2-SCG is made of 4 classes and 7 transitions, detailed in Table 2.

In the second column of Table 2, only relevant variables are represented: X_1, X_2, O_1 and I_1. In particular, the variable $1s/X_2$ is obviously always false in a stable state. The

Table 2. *L2-SCG classes and transitions of the example Fig. 5(b).*

Classes	Some variables at true value	Transitions
C_1	X_1	$\xrightarrow{\{t_\infty, I\_1\_to\_true\}} C_5$
C_5	$X_2,\ O_1,\ I_1$	$\xrightarrow{\{1s\_X\_2\_true\}} C_1$
		$\xrightarrow{\{t_\infty, I\_1\_to\_false\}} C_7$
C_7	$X_2,\ O_1$	$\xrightarrow{\{1s\_X\_2\_true\}} C_1$
		$\xrightarrow{\{t_\infty, I\_1\_to\_true\}} C_9$
C_9	$X_2,\ O_1,\ I_1$	$\xrightarrow{\{1s\_X\_2\_true\}} C_1$
		$\xrightarrow{\{t_\infty, I\_1\_to\_false\}} C_7$

domains of the stable state classes are such that all firings get the clock interval $[0, 0]$, except transition $1s\_X\_2\_true$ with clock interval $[0, 1]$ in the classes C_7 and C_9.

5.5. Consistency check of an update stage

Basing on the equivalence of behaviour between a grafcet and the corresponding ε-TPN, consistency check is only reserved for an update stage where ambiguity in a grafcet model may lead to contradictory actions. Indeed, incompatible update firings may concern stored actions on activation or deactivation:

– stored action on boolean output (or boolean internal variable): set and reset a given variable from a same situation. Subsection 3.4 deals with this case by preventing set and reset of a variable B_i in the same update stage: places Set_B_i and $Reset\_B\_i$ will remain marked at the end of the stage;

– on nonnegative counter: reset, and increasing or decreasing a given counter C_i from a same situation.

To detect such a contradiction after an update stage, a solution may be to make sure that activate or deactivate firings to update the states of various steps (for a step i, it is done by transitions $Deactivate\_Step\_i$ and $Activate\_Step\_i$ in the related ε-TPN) do not cause contradictory stored actions, by pairs of such update transitions. To implement this solution, from the structure of the ε-TPN, the set of update transitions for each variable (counter or boolean) may be established, before computing the state-space. For n activate or deactivate firings concerning the same variable, $n(n-1)/2$ pairs are possible. So, for m variables in the model related to some stored action and n_{max} maximum number of update firings per variable, the complexity of an update stage test is in polynomial order: it is bounded by $m \times n_{max} \times (n_{max} - 1)/2$.

The test fails when for some variable, two update firings are incompatible. However, the presented solution just gives a sufficient condition to detect an ambiguity in stored actions. For instance, multiple updates of a counter in a stage may always lead to the same state.

5.6. Taking into account CTL^* model checking

With ε-TPN, a specific way for computing SCG is necessary to take into account CTL^* model checking, like for the classic TPNs in general. This is justified by the transition t_∞ introducing convex domains, with multiple possible values of clocks for a same firing in a class.

ASCG is computed from an intermediate SCG [5] (SSCG or CSCG), which may be one of the levels of abstraction presented previously. Obviously, the operation of splitting non-atomic state classes is worth doing only when no priority transition is fireable, that is from the classes reached by the in-stability stages: a head transition from such classes are in the set $\{t_\infty\} \cup T_T$. So, L0-SCG, L1-SCG and L2-SCG (in fact L0-SSCG, L1-SSCG and L2-SSCG respectively) will serve to create respectively L0-ASCG, L1-ASCG and L2-ASCG, using the algorithm described in the work [5]. No particular adaptation is necessary for that purpose, a firing t_f from a class to split is just now a set including an external event and a zero delay transition.

As in the work [5], to reduce the size of the intermediate abstraction from which an ASCG is computed, comparing a new class with the stored classes will consist in inclusion test (to obtain CSCG) instead of equality test when computing an SSCG: it is again worth only from the classes reached by the in-stability stages.

The relaxation operation of a class when computing an SSCG or CSCG is justified when an unbounded delay transition (such as t_∞) may remain persistent after firing a delayed transition (such as an element of T_T). So if the grafcet does not contain a non-zero delay transition ($T_T = \emptyset$), relaxation is always useless. The feature 4.3.8 makes anyway useless the relaxation operation, except when observers are introduced in the next section.

6. Conclusion

The formal syntax and semantics for ε-TPN are proposed here, with some modifications made on the original description done in a previous work [12], mainly in order to simplify the construction of the ε-TPN state-space abstractions. Then, two algorithms are described to produce state-space abstractions, according to a partial-order semantics to cope with state explosion due to the concurrency inherent to a TPN formalism. A proposed consistency check should be applied to a stage (i.e. transition firings done in the same instant), especially for the update stages which may contain contradictory orders by the grafcet stored actions. The second algorithm generates the two abstractions relevant from a grafcet point of view: the all situations state-space and the stable situations state-space.

Based on the more clear definitions and characterisations about ε-TPN, a general perspective is to take into account the modelling of other concepts of the GRAFCET standard such as the hierarchical concepts of forcing and enclosure, by extending the translation rules.

Another extension of the current work is to allow model-checking for quantitative time properties. Indeed, TCTL [1] is a quantitative time temporal logic applicable to TPNs [4], and a perspective may be to adapt it to ε-TPN.

7. References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993.

[2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Software Eng.*, 17(3):259–273, 1991.

[3] B. Berthomieu and F. Vernadat. State class constructions for branching analysis of time Petri nets. In *TACAS*, pages 442–457, 2003.

[4] H. Boucheneb, G. Gardey, and O. H. Roux. TCTL model checking of time petri nets. *Journal of Logic and Computation*, 19(6):1509–1540, Dec 2009.

[5] H. Boucheneb and R. Hadjidj. CTL* model checking for time Petri nets. *Theor. Comput. Sci.*, 353(1):208–227, 2006.

[6] G. Bucci and E. Vicario. Compositional validation of time-critical systems using communicating time Petri nets. *IEEE Trans. Softw. Eng.*, 21(12):969–992, 1995.

[7] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[8] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[9] IEC 60848. Grafcet specification language for sequential function charts. Technical report, International Electrotechnical Commission, 2013.

[10] IEC 61131-3. Programmable controllers - part 3: Programming languages. Technical report, International Electrotechnical Commission, 2013.

[11] D. L'Her, P. Le Parc, and L. Marcé. Proving sequential function chart programs using timed automata. *Theoretical Computer Science*, 267(1-2):141–155, 2001.

[12] M. Sogbohossou and A. Vianou. Formal modeling of grafcets with Time Petri nets. *IEEE Transactions on Control Systems Technology*, 23(5):1978–1985, Sept 2015.

[13] N. Wightkin, U. Buy, and H. Darabi. Formal modeling of Sequential function Charts with Time Petri nets. *IEEE Transactions on Control System Technology*, 19(2):455–464, 2011.