Model-checking on grafcets through translation into time Petri net

Médésu Sogbohossou[1] — Rodrigue Yehouessi[1] — Bernard Berthomieu[2]

[1]Département Génie Informatique et Télécommunications
École Polytechnique d'Abomey-Calavi (EPAC), 01 BP 2009 Cotonou, BENIN
medesu.sogbohossou@epac.uac.bj, yehouessi_rodrigue@yahoo.fr

[2]LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, FRANCE
Bernard.Berthomieu@laas.fr

ABSTRACT. The conception of a critical automated system goes through its formal specification in order to proceed to its validation. One of the well-known formalisms to specify the behaviour of such a system is the GRAFCET standard (IEC 60848). GRAFCET being just a semi-formal language, we choose to use an intermediate language to translate without any ambiguity a grafcet model: the time Petri nets (TPN), which take into account quantitative time in a model. In this paper, we propose some verification formulas on GRAFCET charts, via the generated intermediate TPN model: CTL and SE-LTL temporal logics are used to express properties being about situations and actions of the GRAFCET chart. Then, we provide a procedure of implementation by using JGrafchart (a grafcet editor) and the model-checkers in TINA software, namely SELT (for SE-LTL properties) and MUSE (for CTL properties).

RÉSUMÉ. La conception d'un système automatisé critique passe par sa spécification formelle afin de procéder à sa validation. Un des formalismes répandus pour spécifier le comportement d'un tel système est la norme GRAFCET (IEC 60848). GRAFCET n'étant qu'un langage semi-formel, nous choisissons de passer par un langage intermédiaire vers lequel le modèle est traduit sans ambiguité: les réseaux de Petri temporels (TPN), qui prennent en compte le temps quantitatif dans un modèle. Dans cet article, nous proposons des formules de vérification sur les grafcets, via le modèle TPN intermédiaire généré: les logiques temporelles CTL et SE-LTL sont utilisées pour exprimer des propriétés portant sur les situations et les actions du diagramme grafcet. Ensuite, nous proposons une procédure de mise en œuvre passant par l'éditeur de grafcet JGrafchart et les model-checkers du logiciel TINA, à savoir SELT (pour les propriétés SE-LTL) et MUSE (pour les propriétés CTL).

KEYWORDS : grafcet (IEC 60848), Time Petri Net (TPN), model-checking, CTL, SE-LTL

MOTS-CLÉS : grafcet (IEC 60848), réseau de Petri temporel, model-checking, CTL, SE-LTL

1. Introduction

The formal verification of the automated systems [9] is essential before their realization because they are often critical systems and require important costs. There are several techniques for checking such discrete event systems: theorem proof, test, simulation and model-checking [8]. Model-checking is a computer-assisted method for the analysis of systems that can be modeled by state-transition formalisms. The model-checking software takes as input the model (an automaton) and the property to check on it. When a property is not satisfied for the studied system, model-checking may provide a counter-example.

The GRAFCET [1] formalism [11] is widely used by the automation specialists to describe the behavior of the sequential control part of an automated system, by the means of charts in the system specification phase. This standard should not be confused with the SFC one [12, 13] intended for implementation purposes.

However, GRAFCET (and SFC) formalisms are not mathematically defined, and so they contain some ambiguities to clarify via a translation of the chart into a formal representation, such as SMV textual language [15, 1, 14], timed automata [10] or time Petri nets (TPN) [16]. TPNs are structurally (and historically) closer to the GRAFCET than the other formal models.

Thus, the novelty of the present work is to propose a procedure for doing model-checking on a GRAFCET chart (or *grafcet* for short) translated into time Petri net (TPN) according to [16]. Based on the state space construction of a classic TPN [3, 5], two algorithms are implemented to obtain sufficiently compact abstractions which will be the inputs of a model-checker. The first algorithm only preserves informations in the original grafcet (by abstracting extra informations appearing after the translation). The second one enhances the abstraction and displays only states where the grafcet is in a stable situation. Futher, taking into account the specificities of the translation into TPN, some expressions of properties are proposed in CTL and LTL temporal logics, and concern situations and actions of the grafcet. Thanks to SE-LTL [7], it is specially possible to integrate transitions in a property formula.

For the practical experiences, the grafcet editor called JGrafchart [2] is used, and after implementing translation and abstractions, the model-checking is applied by means of two components of TINA software [3] [4], namely SELT (for SE-LTL model-checking) and MUSE (for CTL model-checking).

The remainder of this article is organized as follows. Section 2 shortly presents the used modeling formalisms, and introduces CTL and SE-LTL model-checking fragments. In Section 3 a set of formulas is proposed about the situations and actions of a grafcet. Section 4 describes the different practical steps to achieve model-checking of a grafcet, and contains a case study to illustrate our approach. Finally, Section 5 concludes this paper and gives some outlooks.

1. Acronym in French: *GRAphe Fonctionnel de Commande Etape Transition.*
2. JGrafchart, `http://www.control.lth.se/Research/tools/grafchart.html`
3. TIme petri Net Analyzer (TINA), `http://projects.laas.fr/tina`

2. Modeling formalisms and model-checking

2.1. GRAFCET charts

A GRAFCET chart [11] is a graphical representation modelling the behavior of the control part of an automated system. This representation consists of two parts:

– the *structure* describes the possible evolutions between the situations. It consists of the following basic elements: step, transition and directed link. A situation is the set of active steps at a given instant;

– the *interpretation* enables the relationship between the literal variables (inputs, outputs, delays, internal variables, ...) and the structure. It is done through the transition conditions (containing inputs, rising/falling edges of boolean inputs, delays, ...) and the actions (continuous actions, stored actions).

Figure 2 in Annex B shows an example of a grafcet edited with JGrafchart. It should be noticed that JGrafchart respects only partially the syntax (and the semantics) of the GRAFCET standard. For instance, a continuous action and a stored action on activation are defined respectively with qualifiers N and S (like in the SFC standard), and a timed variable T_j/X_i on a step i (with the value T_j in the second unit) is denoted by $S_i.s > T_j$.

2.2. Translation of grafcet into Time Petri net

Definition 1. *A* Time Petri Net *(TPN) [16] is a tuple* $(P, T, W, W_I, W_R, \downarrow SI, \uparrow SI, M_0)$ *such as:*

– *the nodes:* P *is the set of* places *and* T *is the set of* transitions *(*$P \cap T = \emptyset$*);*

– $W : P \times T \cup T \times P \longrightarrow \mathbb{N}$ *defines the* regular arcs *between nodes (and their weights);*

– $W_R : P \times T \longrightarrow \mathbb{N}$ *defines the* read arcs*;*

– $W_I : P \times T \longrightarrow \mathbb{N}^+ \cup \{\infty\}$ *defines the* inhibitor arcs*;*

– $\downarrow SI : T \longrightarrow \mathbb{Q}^+$ *(resp.* $\uparrow SI : T \longrightarrow \mathbb{Q}^+ \cup \{\infty\}$*) defines the lower (resp. upper) bound of the static interval of the transitions;*

– *the initial marking* $M_0 : P \longrightarrow \mathbb{N}$.

A marking M may enable some transitions in the set T. A transition firing is also conditionned by time information of all the enabled transitions, depending on their static intervals. A firing sequence expresses a behaviour of the modelled system. The standard semantics is used here and is more precisely recalled in a reference such as [6].

The works [16] have proposed a procedure of translating a grafcet into a TPN model, of which syntax is extended by ε infinitesimal delays as bounds on some transitions, allowing to simulate the synchronous semantics of GRAFCET. A extra module (called *phase sequencer*) is necessary to allow a transient evolution without modification of inputs as external events: it forces alternation between the reaction phase (called *evolution* with grafcets) and an external event production (an input change or some timed variable becomes true) in a stable situation. After adding this first module, the generation of the complete TPN is done by translating sequentially: the steps, the inputs, the timed variables, the outputs, the counter variables, the continuous and conditional actions, the stored actions and the grafcet transitions. These grafcet elements (steps, transitions, input variables, actions, ...) correspond to different but connected blocks in the resulting TPN.

The spatial complexity of the translation is polynomial with the number of nodes (steps and transitions), variables or literal terms of the grafcet.

2.3. Model-checking

A model-checking software takes as input an abstraction of the system behavior (a transition system such as a TPN state space [5]) and a property (expresses in Temporal Logic [8]) to check on the model, and answers if the abstraction satisfies or not this property. There are several types of temporal logic including : LTL (Linear Temporal Logic) to express properties on each path of the transition system and CTL (Computational Tree Logic) to express properties taking into account the branching of the different possible futures of the transition system.

A property p is formulated by means of a logical proposition (or formula), of which interpretation (i.e. true or false value) depends on a model $\mathcal{M}$ on which this property is expressed. Thus, the property p verified for the model $\mathcal{M}$ is denoted: $\mathcal{M} \models p$. For temporal properties about a discrete event system, the commonly used model is called *labeled Kripke structure* (LKS): it is a kind of state graph of which each state is labeled with some atomic propositions (in a set AP) true in this state; a transition between two states is labeled by a subset A of events in Σ. In our context, the model $\mathcal{M}$ is the state class graph (SCG) [4] obtained from the translation of a grafcet into an equivalent TPN [16], and the propositions concerns the marking of the different places in the TPN. Here, CTL and LTL temporal logics are used to express properties about situations and actions of the GRAFCET chart.

A path $\pi = (s_0, A_0, s_1, A_1, s_2, A_2, ...)$ of a LKS is an alternating infinite sequence of states ($s_0, s_1, ...$ with s_0 the initial state) and events ($A_0, A_1, ...$ with A_i a set of TPN firings from the state s_i). Notation π^i stands for the suffix of π starting in the state s_i.

The syntax of SE-LTL (State-Event LTL [7]) path formula is given by (where p ranges over AP and a ranges over Σ) :

$$\varphi := p \mid a \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\,\varphi \mid \mathbf{F}\,\varphi \mid \mathbf{G}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi$$

For the SE-LTL semantics, a path-satisfaction of formulas is defined inductively as follows ($\mathcal{L}(s_0)$ is a subset of AP labeling s_0):

1) $\pi \models p$ iff $p \in \mathcal{L}(s_0)$, and $\pi \models a$ iff $a \in A_0$,
2) $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$,
3) $\pi \models \varphi_1 \vee \varphi_2$ iff $\pi \models \varphi_1$ or $\pi \models \varphi_2$,
4) $\pi \models \varphi_1 \wedge \varphi_2$ iff $\pi \models \varphi_1$ and $\pi \models \varphi_2$,
5) $\pi \models \mathbf{X}\,\varphi$ iff $\pi^1 \models \varphi$,
6) $\pi \models \mathbf{F}\,\varphi$ iff $\exists k \geq 0$ s.t. $\pi^k \models \varphi$,
7) $\pi \models \mathbf{G}\,\varphi$ iff $\forall k \geq 0$, $\pi^k \models \varphi$,
8) $\pi \models \varphi_1\,\mathbf{U}\,\varphi_2$ iff $\exists k \geq 0$ s.t. $\pi^k \models \varphi_2$ and $\forall\, 0 \leq j < k$, $\pi^j \models \varphi_1$.

LTL is the restriction of SE-LTL without labels on transitions (i.e. just State LTL). Here, CTL does not consider events, like simple LTL.

The syntax of CTL state formula is given by (φ is a path sub-formula):

$$\phi := p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathbf{E}\,\varphi \mid \mathbf{A}\,\varphi$$
$$\varphi := \mathbf{X}\,\phi \mid \mathbf{F}\,\phi \mid \mathbf{G}\,\phi \mid \phi\,\mathbf{U}\,\phi$$

For the CTL semantics, a state-satisfaction of formulas is defined inductively as follows:

1) $s_0 \models p$ iff $p \in \mathcal{L}(s_0)$,

2) $s_0 \models \neg\phi$ iff $s_0 \not\models \phi$,

3) $s_0 \models \phi_1 \vee \phi_2$ iff $s_0 \models \phi_1$ or $s_0 \models \phi_2$,

4) $s_0 \models \phi_1 \wedge \phi_2$ iff $s_0 \models \phi_1$ and $s_0 \models \phi_2$,

5) $s_0 \models$ **E** φ iff $\exists\pi = (s_0, A_0, \dots)$ s.t. $\pi \models \varphi$ for φ as a path sub-formula,

6) $s_0 \models$ **A** φ iff $\forall\pi = (s_0, A_0, \dots)$, $\pi \models \varphi$ for φ as a path sub-formula.

A path sub-formula φ in CTL is only in the form of: **X** ϕ, **F** ϕ, **G** ϕ or ϕ_1 **U** ϕ_2, according to the same semantics of LTL (items 5-8), and where ϕ, ϕ_1 and ϕ_2 are state sub-formulas.

3. Model-checking on grafcets

We distinguish the properties according to the objects of the grafcet that they handle : situations or actions.

Some properties depend on the specificities of the approach of translation proposed in [16]. Some elements in the resulting TPN report the evolution states and the stability states of a grafcet: it is the case of the places called $Stable$ and $Evolution$, and the transition $Evolution\_end$.

3.1. Properties on the structural aspect

Let S be the set of steps of the considered grafcet.

The LTL properties on the structural aspect are the followings:

1) To find out whether the step X_i is permanently active: **G** X_i

2) To test the existence of a step X_i permanently active: $\bigvee_{X_i \in S}$ **G** X_i

3) To verify that a step X_i is never active: **G** $\neg X_i$

The CTL properties about the structural aspect are the followings:

1) To find out whether the step X_i is permanently active: **AG** X_i

2) To test the existence of a step X_i permanently active: $\bigvee_{X_i \in S}$ **AG** X_i

3) To verify that a step X_i is never active: **AG** $\neg X_i$

4) To know if in the future the step X_i could be permanently active: **EF EG** X_i

5) To test the existence of any step active permanently in the future: $\bigvee_{X_i \in E}$ **EF EG** X_i

6) To check[4] whether the activity of the step X_j is reachable since the one of the step X_i: **AG** $(X_i \Rightarrow$ **AF** $X_j)$

7) To check whether active step X_k is reachable from active step X_i through the active step X_j: **EF** $(X_i \Rightarrow$ **EF** $(X_j \wedge (X_j \Rightarrow$ **EF** $X_k)))$

8) To check that it is possible to find a grafcet execution where the steps X_i, ..., X_n are activated simultaneously: **EF** $((\neg X_i \wedge ... \wedge \neg X_n) \wedge$ **EX** $(X_i \wedge ... \wedge X_n))$

9) To check whether it is possible to return to step X_i or to verify that a step X_i is accessible from all grafcet situations: **AG EF** X_i

4. $\phi \Rightarrow \varphi$ is equivalent to $\neg\phi \vee \varphi$.

10) To check if there is a grafcet situation where there is a deadlock (that is to say a situation that can no longer be left): **EF EG** $(Evolution \Rightarrow Evolution\_end)$

11) To check if there may be total instability in the system: **EF EG** $\neg Evolution\_end$

In fact, the two last properties are not valid in CTL since $Evolution\_end$ is an event. To make such properties valid in a State-Event CTL such as UCTL [2], any classical proposition $Prop$ only made with events (i.e. transition firings) should be replaced by $\mathbf{AX}_{Prop}$ **true**; so, $Evolution\_end$ will become here $\mathbf{AX}_{Evolution\_end}$ **true**.

3.2. Properties on the actions

Let S_j be the set of steps associated with the action $action_j$, T_{j_1} (resp. T_{j_2}) the set of succeeding (resp. preceding) transitions of the steps associated with the action $action_j$. The possible forms of $action_j$ are:

- $action_j$ is a continuous action: $(\bigvee_{X_i \in S_j} X_i) \wedge Stable$
- $action_j$ is a conditioned action by $condition_j$ (a logical expression): $(\bigvee_{X_i \in S_j} X_i) \wedge Stable \wedge condition_j$
- $action_j$ is a stored action (translatable into $\mathbf{AX}_{action_j}$ **true** in UCTL):
  - on activation : $\bigvee_{tr_i \in T_{j_2}} tr_i$
  - on deactivation : $\bigvee_{tr_i \in T_{j_1}} tr_i$

These different forms are used to check the following LTL and CTL properties :

1) LTL property : to show that an action $action_1$ always follows an action $action_2$: $action_2 \Rightarrow \mathbf{F}\ action_1$

2) CTL properties:

a) To show that an action $action_1$ always follows an action $action_2$: **AG** $(action_2 \Rightarrow$ **AF** $action_1)$

b) To show that an action $action_1$ is launched simultaneously with an action $action_2$: **EF** $((\neg action_1 \wedge \neg action_2) \Rightarrow$ **EX** $(action_1 \wedge action_2))$

Naturally, some more general property may mix up both action and step propositions.

4. Implementation of the model-checking

4.1. Procedure

To make model-checking on the grafcet, we proceed as follows:

1) The grafcet to be verified is edited under the JGrafchart software (as shown the figure 2 of the case study in Appendix B). This software generates an XML file containing information on the elements of the grafcet;

2) From the XML file, our Java program generates a **.net** extension file containing information about the elements of the TPN equivalent to the edited grafcet;

3) The implementation of the algorithms 1 and 2 (Annex A) allows us to obtain respectively from the file **.net**, a file **.aut** containing the information on the elements of

the automaton (with unstable and stable states) of the grafcet and another one containing only the information on the stable states of the grafcet (by disregarding unstable states);

4) The TINA **ktzio** tool takes the **.aut** file as input to generate the Kripke structure (**.ktz** extension file on which the verifications are made);

5) Finally, the tools SELT and MUSE (examples in Appendix C) of TINA are used to check the LTL and CTL properties on the grafcet from the Kripke structure.

4.2. Application

The illustration is based on the grafcet as shown in Figure 2 (Annex B). This grafcet models two traffic lights located respectively on a track A and a track B. It contains a transient mode (orange lights blink three times) and a steady state. From the JGrafchart XML file, we generated the equivalent TPN and the two automata of figures 3 and 4 (in Annex B, edited from the generated **.aut** files).

The following examples of properties are checked on the grafcet.

Verification of a LTL property with SELT tool:

- The system leaves the transient mode (firing of transition 13): TRUE. The result of this verification[5] is shown in Figure 1.

```
C:\tina-3.4.4\bin>selt feuxTricolore.ktz
Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 42 states, 42 transitions
0.000s

- [] (tr_13 => () (S0 /\ S4));
TRUE
0.016s
```

Figure 1. *Verification of the exit from the transient mode on the first automaton (with stable and unstable states).*

Verification of some CTL properties with MUSE tool:

- The street A light can stay permanently green: FALSE. Cf. Figure 5 (Annex C).
- The counter that allows blinking of the orange lights in the transient mode can reach the value 4: FALSE. Cf. Figure 6 (Annex C).
- Lights can become green or orange, simultaneously for streets A and B: FALSE. Cf. Figure 7.
- The same lights can pass simultaneously to two different colors (green and red for example): FALSE. Cf. Figure 8.

5. Conclusion

Through these works, we have shown the possibility to check properties (SE-LTL and CTL respectively with the tools SELT and MUSE of the software TINA) on a grafcet after translating it into an equivalent TPN, and subsequently into an automaton representing the state-space. This automaton is as compact as possible by abstracting much information in the TPN and by avoiding multiple interleavings due to the concurrent firings in the TPN.

5. With SELT, operators **G**, **X** and $\wedge$ are denoted resp. [ ], ( ) and /\.

Contrary to the grafcet translation into Timed Automata [10] or TSMV [14], TPNs do not allow model-checking on quantitative time properties with TCTL logic. To introduce timed properties, a perspective to our approach is to integrate observers into the TPN of the translation, to take into account delay events while model-checking the grafcet. Another perspective is the creation of a software implementing all steps of our approach: from editing a grafcet (in full conformity with the IEC60848 standard) until the verification phase. Finally, the extension of CTL to Action/State-Based Temporal Logic UCTL [2] will be an asset to generalize the expression of some properties including events of firing.

6. References

[1] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg. Verification of PLC programs given as sequential function charts. In *Integration of Software Specification Techniques for Applications in Engineering*, pages 517–540, 2004.

[2] M. H. Ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications. In *FMICS*, pages 133–148. Springer, 2007.

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

[4] B. Berthomieu and F. Vernadat. Time Petri nets analysis with TINA. In *Quantitative Evaluation of Systems, QEST 2006.*, pages 123–124. IEEE, 2006.

[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] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *IFM*, vol. 2999, pages 128–147. Springer, 2004.

[8] E. M. Clarke, T. A. Henzinger, and H. Veith. *Introduction to Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018.

[9] D. Darvas, I. Majzik, and E. B. Viñuela. PLC program translation for verification purposes. *Periodica Polytechnica, Electrical Engineering and Computer Science*, 61:151–165, 2017.

[10] 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.

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

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

[13] A. Karatkevich. *Petri Nets in Design of Control Algorithms*, pages 1–14. Springer International Publishing, Cham, 2016.

[14] N. Markey and P. Schnoebelen. TSMV: A symbolic model checker for quantitative analysis of systems. In *QEST*, vol. 4, pages 330–331, 2004.

[15] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver. An overview of model checking practices on verification of PLC software. *Softw. Syst. Model.*, 15(4):937–960, October 2016.

[16] 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.

A. Algorithms

Two algorithms are proposed and implemented (in Java) to obtain sufficiently compact abstractions.

Algorithm 1. *SCG (LTL): first abstraction*

```
Save and stack (LIFO) the initial state;
while (the Stack is not empty) do
    Unstack a state (or state class);
    if (a grafcet transition is fireable) then
        Fire all simultaneously fireable grafcet transitions;
        Fire all fireable transitions for modifying literals;
        if (the last reached state is new) then Save and stack it;
    else if (Evolution_End is fireable) then
        Fire transition Evolution_End;
        Fire all fireable transitions for continuous actions;
        if (the last reached state is new) then Save and stack it;
    else if (Change_input or a delay transition of some timed variable model are fireable) then
        for each fireable transition do
            Fire all fireable transitions until a grafcet transition or Evolution_End is fireable;
            if (the last reached state is new) then Save and stack it;
        end
    end
end
```

Algorithm 2. *SCG (LTL): second abstraction*

```
Stack (LIFO) the initial state;
while (the Stack is not empty) do
    Unstack a state (or state class);
    if (a grafcet transition is fireable) then
        Fire all simultaneously fireable grafcet transitions;
        Fire all fireable transitions for modifying literals;
        if (the last reached state is new) then Stack it;
    else if (Evolution_End is fireable) then
        Fire transition Evolution_End;
        Fire all fireable transitions for continuous actions;
        if (the last reached state is new) then Stack it;
    else if (Change_input or a delay transition of some timed variable model are fireable) then
        for each fireable transition do
            Fire all fireable transitions until a grafcet transition or Evolution_End is fireable;
            if (the last reached state is new) then Save and stack it;
        end
    end
end
```

Algorithm 1 only preserves informations in the original grafcet, by abstracting extra informations appearing after the translation: for instance, firings related to the synchronous updatings (step states and literal variables) are abstracted.

Algorithm 2 is the same as Algorithm 1, except that only state classes corresponding to the stable situations of the grafcet (line 15) are saved, and only $Change\_input$ and delay transition firings from theses classes are displayed.

The two algorithms assume that all possible interleavings of firings which symbolize a grafcet evolution between two stable situations lead to the same global state. This assumption (and other prerequisites) were discussed in [16].

B. The case study

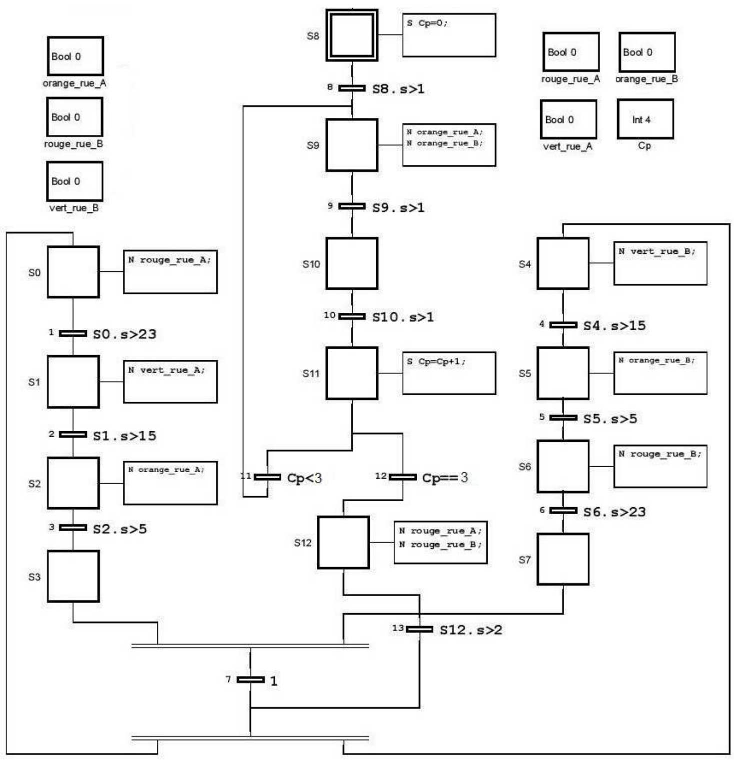

Figure 2. *Case study*

From the JGrafchart XML file of Figure 2, we have generated the equivalent TPN and the two automata of figures 3 and 4 (edited graphically with the tool ND of TINA taking as input the generated **.aut** files). To summarize:

– the TPN obtained is made of 75 places, 96 transitions, 205 regular arcs, 110 read arcs and 20 inhibitor arcs;

– the first abstraction (figure 3) is made of 42 states and 42 transitions;

– the second abstraction (figure 4) is made of 12 states and 12 transitions.

Figure 3. *Automaton based on algorithm 1 (stable and unstable states of the grafcet)*

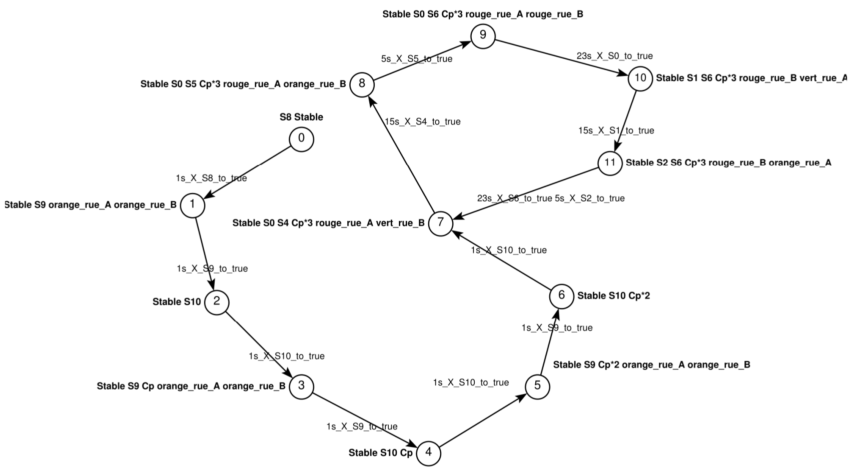

Figure 4. *Automaton based on algorithm 2 (stable states of the grafcet)*

C. Some TINA results of the case study

These results concern CTL properties tested with MUSE tool, only by using the second automaton (with only stable states).

```
C:\tina-3.4.4\bin>muse feuxTricolore2.ktz -prelude ctl.mmc -b
Muse version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 12 states, 12 transitions
0.000s
- EF EG vert_rue_A;
it : states
FALSE
0.000s
```

Figure 5. *Checking the permanent activation of the green light of the street A*

```
C:\tina-3.4.4\bin>muse feuxTricolore2.ktz -prelude ctl.mmc -b
Muse version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 12 states, 12 transitions
0.000s
- EF (Cp=4);
it : states
FALSE
0.000s
```

Figure 6. *Checking the state of the counter*

```
C:\tina-3.4.4\bin>muse feuxTricolore2.ktz -prelude ctl.mmc -b
Muse version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 12 states, 12 transitions
0.016s
- EF ((vert_rue_A /\ vert_rue_B) \/ (vert_rue_A /\ orange_rue_B) \/ (vert_rue_B
/\ orange_rue_A) \/ (S1 /\ S5));
it : states
FALSE
0.000s
```

Figure 7. *Checking a safety property*

```
C:\tina-3.4.4\bin>muse feuxTricolore2.ktz -prelude ctl.mmc -b
Muse version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 12 states, 12 transitions
0.016s
- EF ((vert_rue_A /\ rouge_rue_A) \/ (vert_rue_A /\ orange_rue_A) \/ (rouge_rue_
A /\ orange_rue_A));
it : states
FALSE
0.000s
```

Figure 8. *Checking no two lights on at the same place*

Fig. 5 shows that a green light will never stay indefinitely turned on. Fig. 6 shows that the orange lights in the transient mode will not blink more than three times. Fig. 7 displays that the lights for the two crossing streets will never allow all road users to pass through simultaneously. Finally, Fig. 8 shows that two lights may not turn on simultaneously for some user.

Adjustment module

to give auto-adaptiveness behavior to flood forecasting systems

Joel TANZOUAK*, Idrissa SARR+, Blaise YENKE*, Ndiouma BAME+, Serigne FAYE+

*The University of Ngaoundere (Cameroon),
+The University of Cheikh Anta Diop (Senegal)

RÉSUMÉ. La prévision est aujourd'hui un facteur clé dans la minimisation des dégâts causés par les inondations. En effet, les systèmes de prévision d'inondations (FFS) fonctionnent pour la plupart dans les pays developpés et utilisent des modèles hydrauliques pour fournir des prévisions du niveau et/ou du débit des rivières en se basant sur les prévisions meteo (NWP). Ces données de prévision sont utilisées pour fournir des alertes d'inondations; Il est donc important d'utiliser de bons modèles hydrauliques pour obtenir des données précises. De nombreux modèles hydrauliques ont été construits pour les FFS. Cependant, la différence entre les paramètres environnementaux et climatologiques entre les régions rend très difficile l'utilisation de ces FFS dans d'autres régions. De plus, l'évolution constante au cours du temps de l'environnement, causée par des facteurs anthropiques, nécessite un processus de recallage fréquent des modèles hydrauliques pour qu'ils s'ariment aux changements environnementaux. Par conséquent, il est nécessaire de construire des FFS qui s'adaptent dynamiquement aux changements environnementaux sans processus de recallage. L'objectif de cet article est de proposer une extension des FFS en introduisant un module d'ajustement qui utilise des données collectées en temps réel à partir de réseaux de capteurs combinées avec des données prévisionnelles issues des modèles hydrauliques, pour donner une capacité d'auto-adaptation dynamique aux FFS. Les résultats obtenus à partir d'expériences empiriques montrent les avantages de notre mécanisme d'ajustement dans l'auto-adaptation des FFS.

ABSTRACT. Forecasting is now a key factor in minimizing the damages caused by Flood. Indeed, Flood forecasting systems (FFS) operate mostly in developed countries and use hydraulic models to provide forecasts of river levels and / or flow, based on numeric weather predictions (NWP). These forecast data are used to provide flood alerts, so it is therefore important to use good hydraulic models to obtain accurate flood forecast. Many hydraulic models have been built for FFS. However, the difference between environmental and climatological parameters between regions makes very difficult the use of these FFS in other regions. Moreover, the constant evolution over time of the environment, caused by anthropic factors, need a frequent process updates of hydraulic models so that they can be adapted to environmental changes. Therefore, it is necessary to build FFS that dynamically adapt to environmental changes without a recall process. The purpose of this article is to propose an extension of FFS by introducing an adjustment module that uses real-time data collected from sensor networks combined with predictive data from hydraulic models, to provide FFS with a dynamic self-adaptation ability. The results obtained from empirical experiments show the advantages of our adjustment mechanism in the self-adaptation of FFS

MOTS-CLÉS : Modèle hydraulique, Réseau de Capteurs, Module d'ajustement, Auto-adaptabilité

KEYWORDS : Hydraulic model,Sensors network, Adjustment module, Auto-adaptiveness