Model-Checking of Concurrent Real-Time Software Using High-Level Colored Time Petri Nets with Stopwatches

Imane Haur, Jean-Luc Béchennec, Olivier H. Roux

Model-checking of concurrent real-time software using High-level Colored Time Petri Nets with stopwatches.

Imane Haur[ab], and Jean-Luc Béchennec[a], and Olivier H. Roux[a]

[a]*Ecole Centrale de Nantes, CNRS, LS2N, 1 Rue de la Noë, Nantes, France.*

[b]*Huawei Paris Research center, 20 Quai du Point du Jour, Boulogne Billancourt, France.*

Imane.Haur@ls2n.fr, Jean-Luc.Bechennec@ls2n.fr, Olivier-h.roux@ec-nantes.fr

The control of real-time systems often requires taking into account simultaneous access in true parallelism to shared resources. This is particularly the case for multicore execution platforms. Timed automata or time Petri nets do not capture these features directly. We first define High-level Colored Time Petri Net (HCTPN) that extends time Petri Nets with color and high-level functionality encompassing both timed multi-enableness of transitions and sequential pseudo code. We then extend HCTPN with stopwatches to allow the modeling of preemptive scheduling, which is an important feature in the real-time context. We prove that the reachability problem is decidable for HCTPN but is undecidable for HCTPN with stopwatches, and we propose an abstraction of the state space for these models. We apply this approach to model a preemptive multi-core real-time application that uses a spinlock mechanism in order to check all possible execution paths, interleaving of service calls, and preemptive scheduling.

Keywords: Multi-core execution; High-level Colored Time Petri Nets with stopwatches; Model-checking

Introduction

Real-time systems are now present in various contexts and applications in our everyday lives. A real-time system interacts with a complex external environment and should meet deadlines, guarantee timing constraints, and consider the correctness of the computation. These systems have evolved and need to use increasingly complex hardware and software architectures to achieve the required level of performance. In addition, the demand for high-performance computing among applications has led to a rise in the usage of multi-core chips.

Implementing multi-core real-time systems requires concurrent access in true parallelism to shared resources. The design of such a system must also be predictable, i.e., its behavior must be expected concerning the time requirements. It is required to implement all of the desired functionalities and validate both functional and temporal accuracy. In order to increase confidence and prevent unexpected behavior, system verification is essential.

Formal approaches ensure system confidence, and the emergence of new software tools has led to their usability. Model-checking (Clarke et al., 2018) is one of the most popular families of formal methods. This technique effectively deals with concurrency and interaction between parallel processes, which are the significant sources of error in

the systems. It relies on the algorithmic exploration of the system model's whole state space to verify the correctness of properties on the entire execution path.

Petri nets, known as a place/transition model, can simulate concurrency. For the lack of data structures, Petri nets are unsuitable for modeling systems where data affects the system's behavior. High-level Petri nets (Hillah et al., 2006) have been proposed for modeling scientific problems with complex structures allowing the description of both system data and control. The term High-level Petri net is then used for many Petri nets (Kindler and Petrucci, 2009) such as Predicate/Transition Nets, colored Petri nets, or hierarchical Petri nets. However, the common point is that they allow the manipulation of different types of expressions that use state variables. Input arcs are labeled with boolean expressions specifying conditions (guards or gates) that can also be associated with transitions. Arc annotations are expressions that can be associated with output arc. They can be viewed as computing systems that operate on shared data.

Scientific contribution. We have mainly three challenges in reasoning about the behavior of concurrent real-time systems: i) The real-time system is susceptible to a variety of stimuli, in particular, real-time stimuli, such as periodic interruptions generated by timers; ii) Applications and code blocks can be executed concurrently on several cores, and iii) Some parts of the code can be executed simultaneously by several cores.

Our work leads us to two main contributions to achieving the presented objectives. We first define High-level Colored Time Petri Net (HCTPN) that extends time Petri Nets with color and high-level functionality encompassing both timed multi-enableness of transitions and sequential pseudo code. We then extend HCTPN with stopwatches to allow the modeling of preemptive scheduling, which is an important feature in the real-time context. We prove that the reachability problem is decidable for HCTPN but is undecidable for HCTPN with stopwatches, and we propose an abstraction of the state space for these models.

We then use the High-Level Colored Time Petri Nets with stopwatches to model a preemptive concurrent real-time application case study. For its verification, we use formal methods, particularly the model-checking technique. We rely on the ROMÉO model-checker tool, improved to support this extended formalism, and available under a free license (Lime et al. (2009)).

Paper outline. The paper is structured as follows. Section 2 presents some related works. Sections 3 and 4 define the High-Level Colored Time Petri Nets (HCTPN) with stopwatches. Section 5 studies reachability problem for HCTPN and HCTPN with stopwatch, presents decidability and undecidability results and propose a state space symbolic abstraction. Section 6 deals with the well-known Path finder mono-core scheduling problem and section 7 describes a case study of a dual-core concurrent preemptive application built with the HCTPN formalism and its verification using the ROMÉO model-checker tool. Section 8 concludes the paper.

Related works on timed models

Time-based models allow the modeling and verification of real-time applications by considering task execution times and synchronization mechanisms. We consider here the time extension of finite timed automata and time Petri nets.

Timed automata. Timed automata (Alur and Dill, 1994) extend finite automata with clocks to consider time. The values of the clocks increase during the execution of timed automata and can be associated with constraints called invariants (Henzinger et al., 1994). The clocks can be reset to 0 as part of the transition update when the latter is fired.

Petri nets and time Petri nets have two main temporal extensions. Time Petri nets (Merlin and Farber, 1976) where each transition is associated with a time interval that specifies the possible firing dates. Timed Petri Nets (Ramchandani, 1974) is a

temporal extension where minimum (or exact) durations represent time. Time can be associated with transitions (T-time), places (P-time), and arcs (A-time). Boyer et al. compare the expressiveness of these three models in (Boyer and Roux, 2008). T-time Petri nets are the most widely used in real-time systems (Berthomieu and Diaz, 1991) and are those extended in this work.

Timed model with stopwatches. Timed models can be extended with stopwatches instead of clocks to model the temporal interruption of actions and subsequent resumption. Several extensions of these models have been proposed to express the suspension and resumption of actions by adding the stopwatch notion. Timed automata are extended with stopwatches by (Cassez and Larsen, 2000). For Time Petri Nets (TPN), several extensions have been proposed to model preemptive real-time tasks: Scheduling-TPN (Roux and Déplanche, 2002; Lime and Roux, 2003), Preemptive-TPN (Bucci et al., 2004), and Time Petri nets with inhibitor hyperarcs (IHTPN) (Roux and Lime, 2004).

All these models allow the modeling of preemptive scheduling. Time Petri nets are well adapted for modeling concurrent real-time systems, but face the problem of modeling true concurrency for simultaneous access to resources such as those encountered in the multi-core context.

In the following, we give the informal definitions of Petri net formalism with its different extensions (time, color, high-level, and stopwatches).

Informal presentation

Petri nets

Petri nets are a mathematical formalism and one of the many modeling languages used to describe distributed concurrent systems. A Petri net is a directed bipartite graph whose vertices are places and transitions. A place can contain any number of tokens. A marking M of a Petri Net is a vector representing the number of tokens of each place. A transition is enabled (it may fire) in M if there are enough tokens in its input places for the consumption to be possible. Firing a transition from a marking M consumes tokens from each of its input places and produces tokens in each of its output places.

High-level Petri nets

Petri nets can be classified into two classes: ordinary Petri nets and high–level Petri nets. High-level Petri nets (Hillah et al. (2006)) are proposed for modeling scientific problems with complex structures and manipulating different types of expressions made up of variables and written in terms of a predefined syntax. In high-level nets, each token can carry complex information which, e. g., may describe the entire state of a process or a database and handle different expressions and data structures.

The precondition (guard) and postcondition (update) over a set of variables (X) are associated with transitions. A transition is enabled (it may fire) if there are enough tokens in its input places and if the guard is true. When the transition fires, the corresponding updates are executed, modifying the values of the variables. The variables take their values in a finite state (such as bounded integer or enumerated type...), guards are boolean expressions over X, and updates can be described as a sequence of imperative code expressed in a programming language but whose execution is atomic from the transition firing point of view.

Colored Petri nets

The colored extension of Petri nets allows the distinction between tokens.

Although the set X of High-level Petri nets presented in the previous paragraph can be of arbitrarily complex type, places in colored Petri nets contain tokens of one type. This type noted C is called the color set of the place.

An arc from a place to a transition (PT) specifies the color(s) that enabled the transition, and its firing will consume it. An arc from a transition to a place (TP) specifies the token color produced in that place by the firing of the transition. A particular color called *any* indicates in a PT arc that any color enabled the transition, and in a TP arc that the color consumed in the input place will be the one produced in the output place.

A marking M of a colored Petri Net represents not only the number of tokens in each place but also their respective colors. That is represented either by a multiset or by a matrix.

Time Petri Nets

Time Petri nets (TPN) extend Petri nets with temporal intervals (such as $[\alpha,\beta]$ or $[\alpha,+\infty[$) associated with transitions, specifying firing delay ranges for each transition. Assuming transition t became last enabled at time d and the endpoints of its firing interval are α and β, then t cannot fire earlier than $d + \alpha$ and must fire no later than $d + \beta$ unless disabled by the firing of another transition. Firing a transition takes no time.

To describe the semantics of TPN, we usually consider that a clock is associated with each transition. This clock is set to zero when the transition is newly enabled, and the transition fires when the value of the clock is in the firing interval.

Colored Time Petri Nets

For real parallelism or with interleaving semantics of timed systems, the notion of multiple enableness is needed. It refers to the fact that a transition is enabled at least twice in the same state, which implies a dynamic number of timers. Multiple enableness in time Petri nets is a natural way for modeling paradigms like multiple servers and multiple instances of codes (Boyer and Diaz (2001)).

For Colored Time Petri Nets, multiple enableness occurs when several combinations of colors enable a transition at a given time. In this case, there can be at most one clock per color and per transition.

Time Petri Nets with stopwatches

Time Petri nets with stopwatches extend TPN by replacing clocks by stopwatches. The time derivative of the stopwatch of a transition is in the set of rates $\{0,1\}$ and is given by a function from Markings. Hence the time associated with a transition can be suspended and later resumed at the same point. Moreover, transition with a 0 time derivative cannot fire.

Since the clocks are replaced by stopwatches, in the case of Colored Time Petri Nets with stopwatches, there is at most one stopwatch per color and per transition.

Formal definition

No Petri net model encompasses both temporal aspects (timed transitions), preemptive scheduling (stopwatches), code handling (high-level functionalities) and true concurrency modelling (multiple enableness of transitions).

The introduction of color in time Petri nets leads to multiple enableness of transitions and then allows the modeling of multiple servers and multiple instances of codes. We then propose a Petri Nets model which encompasses both colors, high-level functionalities and stopwatches. We now give the formal definition.

High-level Colored Time Petri Net

Notations The sets $\mathbb{N}$, $\mathbb{Q}_{\geq 0}$, and $\mathbb{R}_{\geq 0}$ are, respectively, the sets of natural, non-negative rational, and non-negative real numbers. An interval I of $\mathbb{R}_{\geq 0}$ is a $\mathbb{Q}$-interval iff its left endpoint $^{\uparrow}I$ belongs to $\mathbb{Q}_{\geq 0}$ and its right endpoint $I^{\downarrow}$ belongs to $\mathbb{Q}_{\geq 0} \cup \{\infty\}$. We denote by $\mathcal{I}(\mathbb{Q}_{\geq 0})$ the set of $\mathbb{Q}$-intervals of $\mathbb{R}_{\geq 0}$.

B^A stands for the set of mappings from A to B. If A is finite and $|A| = n$, an element of B^A is also a vector in B^n. The usual operators $+, -, <$ and $=$ are used on vectors of A^n with $A = \mathbb{N}, \mathbb{Q}, \mathbb{R}$ and are the point-wise extensions of their counterparts in A.

Definition and semantics

Colored Petri nets allow tokens to have a data value called the token color. In the applications we are considering, the color of a token actually represents the processor on which the code is executed. We therefore consider token of integer type that designates the processor number. Moreover, we add a special color called *any* to specify that any color can be used for enabling and firing a transition.

We consider a set C of colors. An arc is either associated with a color of C or can take on the particular color called *any*. For the firing of a transition, all its arcs associated with the *any* color must match to instantiate *any* at the same color taken from C.

If several values of *any* allow its enabling, the transition is multi-enabled, and in this case, several clocks (one per color) are associated with the transition, allowing several firing dates depending on the enabling date and the time interval.

The formal definition is as follows.

Definition 1 (High-level Colored Time Petri Net). A High-level Colored Time Petri Net *(HCTPN) is a tuple* $\mathcal{N} = (P, T, X, C, \mathsf{pre}, \mathsf{post}, (m_0, x_0), guard, update, I)$ *where*

- P is a finite non-empty set of places,
- T is a finite set of transitions such that $T \cap P = \emptyset$,
- X is a finite set of variables taking their value in the finite set $\mathbb{X}$ (such as bounded integer),
- C is a finite set of colors and $C_{any} = C \cup \{any\}$ where *any* is a variable that can be instantiated to any value of C,
- $\mathsf{pre} : P \times T \to \mathbb{N}^{C any}$ is the backward incidence mapping,
- $\mathsf{post} : P \times T \to \mathbb{N}^{C any}$ is the forward incidence mapping,
- $guard : T \times X \times P \times C_{\bullet} \to \{\text{true,false}\}$ is the guard function with $C_{\bullet} = C \cup \{\bullet\}$ where $\bullet$ denotes the fact that no color is specified,
- $update : T \times X \times P \times C_{\bullet} \to \mathbb{X}^X \times \mathbb{N}^{P \times C}$ is the update function,
- $(m_0, x_0) \in \mathbb{N}^{P \times C} \times \mathbb{X}^X \to$ is the initial values m_0 of the marking and x_0 of the variables,

- $I : T \rightarrow \mathcal{I}(\mathbb{Q}_{\geq 0})$ is the static firing interval function.

Discrete behavior: For a marking $m \in \mathbb{N}^{P \times C}$, $m(p)$ is a vector in $\mathbb{N}^{C}$, and $m(p)[c]$ represents the number of *tokens* of color $c \in C$ in place $p \in P$. A valuation of the set of variables X is noted $x \in \mathbb{X}^{X}$. (m,x) is a discrete state of HCTPN.
Enabling a transition: Informally, an arc is associated either with a color $c \in C$ or with a particular color called *any*. To enable transition t, a place p with an arc from p to t must have enough tokens with the arc's color. Moreover, all the arcs of t associated with *any* must agree on the color given to *any*. Therefore, we forbid an arc to be associated with both *any* and a color $c \in C$.

An arc $\mathsf{pre}(p,t) \in \mathbb{N}^{C any}$ is a vector such that $\mathsf{pre}(p,t)[c]$ is the number of tokens of color $c \in C$ in place p needed to enable the transition t and $\mathsf{pre}(p,t)[any] > 0$ represents the fact that any color can enable the transition. Let $T_{any} \in T$ the set of transitions that can be enabled by *any* color: i.e. $T_{any} = \{t \in T, \exists p \in P,$ s.t. $\mathsf{pre}(p,t)[any] > 0\}$. Moreover, we define the set $T_{\overline{any}} = T \backslash T_{any}$.

A transition $t \in T$ is said to be *enabled* by a given marking $m \in \mathbb{N}^{P \times C}$ in two cases depending on whether $t \in T_{any}$ or not:

- if $t \in T_{any}$, and $\forall p \in P$ and $\forall c \in C$, $m(p)[c] \geq \mathsf{pre}(p,t)[c]$. We denote $\mathsf{en}(m,t) \in \{true,false\}$, the true value of this condition.
- if $t \in T_{any}$, and $\exists c_a \in C$ such that $\forall p \in P$, $m(p)[c_a] \geq \mathsf{pre}(p,t)[any]$ and $\forall c \in C \setminus \{c_a\}$, $m(p)[c] \geq \mathsf{pre}(p,t)[c]$. The corresponding set of color c_a is noted $\mathsf{colorSet}_{\mathsf{any}}(m,t) \subseteq C$

Finally, a transition $t \in T$ is said to be *enabled* by a given marking $m \in \mathbb{N}^{P \times C}$ and a valuation $x \in \mathbb{X}^{X}$ if $\mathsf{en}(m,t) = true$ and either $\mathsf{colorSet}_{\mathsf{any}}(m,t) = \emptyset$ and $guard(m,t,x,\bullet) = true$ or $\exists c_a \in \mathsf{colorSet}_{\mathsf{any}}(m,t) \neq \emptyset$ and $guard(m,t,x,c_a) = true$.

We illustrate the enabling condition with two examples with two colors $C = \{blue,red\}$. For the HCTPN given in Figure 1.a, the transition $T_1 \in T_{\overline{any}}$.

We have $\mathsf{pre}(T_1) = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{ccc} red & blue & any \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$. The initial marking is $m_0 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \end{array}$ that enables the transition T_1 and $\mathsf{en}(m_0,T_1) = true$.

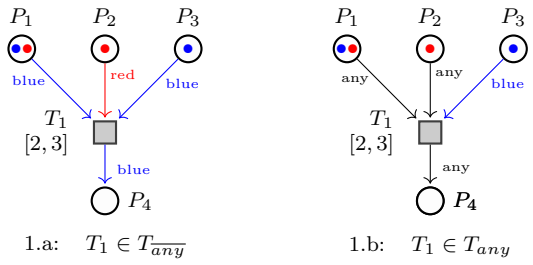

Figure 1. Enabling transition.

Now we consider the HCTPN given in Figure 1.b with the same initial marking m_0 but where the transition $T_1 \in T_{any}$ since at least one arc (here two) is associated with the color *any*.

We have $\mathsf{pre}(T_1) = \begin{matrix} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{matrix} \begin{matrix} red & blue & any \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{matrix}$. The transition is enabled only if *any* takes the *red* value then $\mathsf{colorSet}_{\mathsf{any}}(m_0, T_1) = \{red\}$. If place $P2$ had two tokens with one token per color, then the transition would be multi-enabled by the two colors leading to $\mathsf{colorSet}_{\mathsf{any}}(m_0, T_1) = \{blue, red\}$.

The firing of a transition. An arc $\mathsf{post}(p,t) \in \mathbb{N}^{C_{any}}$ is a vector such that $\mathsf{post}(p,t)[c]$ is the number of tokens of color $c \in C$ produced in place p by the firing of the transition t, and $\mathsf{post}(p,t)[any]$ gives the number of tokens produced in p with the color $c \in \mathsf{colorSet}_{\mathsf{any}}(m,t)$ used for the enabling and then for the firing of t.

Firing an enabled transition $t \in T_{\overline{any}}$ from (m,x) such that $\mathsf{en}(m,t) = true$ and $guard(m,t,x,\bullet) = true$ leads to a new marking m' defined by $\forall c \in C, \forall p \in P$, $m'(p)[c] = m(p)[c] - \mathsf{pre}(p,t)[c] + \mathsf{post}(p,t)[c]$ and a new valuation $x' = update(m,t,x,\bullet)$. This new marking is denoted $m' = \mathsf{firing}(m,t,\bullet)$ where $\bullet$ denotes the fact that no *any* color has to be instantiated for this firing.

We denote by $\mathsf{newen}((m,x),t,c)$ the set of transitions that are newly enabled by the firing of t from (m,x) with the color c ($c = \bullet$ if $t \in T_{\overline{any}}$).

Let us go back to the HCTPN of Figure 1.a, the firing of $T_1 \in T_{\overline{any}}$ from m_0 leads to the marking $m_1 = \begin{matrix} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{matrix} \begin{matrix} red & blue \\ \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \end{matrix}$. It is noted $m_0 \xrightarrow{(T1,\bullet)} m_1$.

Let us now consider the HCTPN of Figure 1.b, the firing of $T_1 \in T_{any}$ is possible only for *any* = *red* and leads to the marking $m_2 = \begin{matrix} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{matrix} \begin{matrix} red & blue \\ \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{pmatrix} \end{matrix}$. It is noted $m_0 \xrightarrow{(T1,red)} m_2$.

If place P_2 had two tokens with one blue and one red color, T_1 is multi-enabled, and the firing of $T_1 \in T_{any}$ is possible for *any* = *red* or *any* = *blue*. For *any* = *blue*, it leads to the following marking m_3 from this new initial marking $m'_0 = \begin{matrix} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{matrix} \begin{matrix} red & blue \\ \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \end{matrix}$

$\xrightarrow{(T1,blue)} m_3 = \begin{matrix} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{matrix} \begin{matrix} red & blue \\ \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \end{matrix}$.

High-level functionalities: We now illustrate the high-level functionalities. In the Figures, the guards are in brown, and the updates are in purple.

The model in Figure 2 is an HCTPN with a set of three colors C = {*blue,red,black*}. Several combinations of color usage, on guards and in updates, via the $*any* variable are presented[1].

```
typedef color {blue = 0, red = 1, black = 2};
int[3] cpt = {2,2,5};

int f(int firedColor, int[3] c) {
        if (firedColor == red) {
                return c[firedColor]*2;
        }
        else if  {
                return c[firedColor] -1 ;
        }
}
```

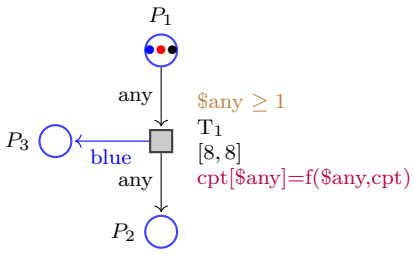

Figure 2. High-level manipulation of variables.

Transition $T_1 \in T_{any}$ since at least one arc is associated with the color *any*. A firing of this transition produces a blue token in P_3 and produce a token in P_2 with the color ($*any*) used for the firing. Moreover, the value of $*any* is used in the precondition (guard) and the postcondition (update). Hence transition T_1 is not enabled by blue token because of the guard $any ≥ 1. Moreover, the firing of T_1 leads to the execution of the update cpt[$any]=f($any,cpt). Then the transition T_1 will be fired twice respectively with a red and a black tokens leading to a marking with a red and a black tokens in P_2 and 2 blue tokens in P_3. It remains a blue token in P_1 and the final value of cpt is {2,4,4}.

Time behavior: For any $t \in T_{any}$, $v(t,c)$ is the valuation of the clock associated with t and the color $c \in C$. i.e., it is the time elapsed since the transition t has been newly enabled by m with $c \in \mathsf{colorSet}_{any}(m,t)$. For other transitions $t \in T_{any}$, $v(t,\bullet)$ is the valuation of the clock associated with t.

$\bar{0}$ is the initial valuation with $\forall t \in T, \forall c \in C \cup \{\bullet\}, \bar{0}(t,c) = 0$.

As an example, if we keep only the useful clocks, the initial valuation of the HCTPN of Figure 1 is $v_0 = T_1\begin{pmatrix} \bullet & red & blue \\ 0 & & \end{pmatrix}$ for Figure 1.a, $v_0 = T_1\begin{pmatrix} \bullet & red & blue \\ 0 & 0 & \end{pmatrix}$ for Figure 1.b, and $v_0 = T_1\begin{pmatrix} \bullet & red & black \\ 0 & & \end{pmatrix}$ for Figure 2.

A *state* of the net $\mathcal{N}$ is a tuple $((m,x),v)$ in $\mathbb{N}^{P\times C} \times \mathbb{X}^X \times {\mathbb{R}_{\geq 0}}^{T\times C}$, where: m is a marking, x is a variable valuation and v is a valuation of the clocks.

Definition 2. (Semantics of a HCTPN). The semantics of a HCTPN is a timed transition system (Q,Q_0,→) where:

---

1 In the example in this section and in the examples that follow we present models designed with the tool Roméo. In this tool, $*any* is used instead of *any* in guards and updates for syntactic reasons but both have the same meaning.

- $Q \subseteq \mathbb{N}^{P\times C} \times \mathbb{X}^{X} \times {\mathbb{R}_{\geq 0}}^{T\times C}$
- $Q_0 = ((m_0,x_0),\bar{0})$
- $\rightarrow \in Q \times ((T \times C \cup \{\bullet\}) \cup \mathbb{R}_{\geq 0}) \times Q$ consists of two types of transitions:
  - discrete transitions (firing t from $((m,x),v)$) iff:
    - $((m,x),v) \xrightarrow{(t\in T_{\overline{any}},\bullet)} ((m',x'),v')$ with
      - $\mathsf{en}(m,t) = true$ and $v(t) \in I(t)$,
      - $m' = \mathrm{firing}(m,t,\bullet)$
    - $((m,x),v) \xrightarrow{(t\in T_{any},c)} ((m',x'),v')$ with
      - $c \in \mathsf{colorSet}_{\mathsf{any}}(m,t)$ *and* $v(t,c) \in I(t)$,
      - $m' = \mathrm{firing}(m,t,c)$
    - $guard(t,x) = true$ *and* $x' = update(t,x)$
    - $\forall t' \in T_{any}$ *s.t.* $\mathsf{en}(m',t') = true$
      - $v'(t',\bullet) = v(t',\bullet)\ if\ t' \notin \mathsf{newen}((m,x),t,\bullet)$,
      - $v'(t',\bullet) = 0$ otherwise.
    - $\forall t' \in T_{any}$ and $\forall c \in \mathsf{colorSet}_{\mathsf{any}}(m',t')$
      - $v'(t',c) = v(t',c)\ if\ t' \notin \mathsf{newen}((m,x),t,c)$,
      - $v'(t',c) = 0$ otherwise.
  - time transitions: $((m,x),v) \xrightarrow{d\in\mathbb{R}_{\geq 0}} ((m,x),v')$ iff :
    - $\forall t \in T_{\overline{any}}$ s.t. $\mathsf{en}(m,t) = true$,
      - $v'(t,\bullet) \leq I(t)\downarrow$
      - $v'(t,\bullet) = v(t,\bullet) + d$
    - $\forall t \in T_{any}$ and $\forall c \in \mathsf{colorSet}_{\mathsf{any}}(m,t)$,
      - $v'(t,c) \leq I(t)\downarrow$
      - $v'(t,c) = v(t,c) + d$

We now illustrate the main features of HCTPN in an example. The guards are in brown in the Figures and the update in purple.

Examples of HCTPN

We give three examples. The first two examples illustrate the high-level functionalities, and the third illustrates the notion of color and multi-enableness.

Example 1: Let's go back to the HCTPN given in Figure 2. The initial marking $m_0 = \begin{matrix} \\ P_1 \\ P_2 \\ P_3 \end{matrix}\begin{matrix} \begin{matrix} blue & red & black \end{matrix} \\ \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{matrix}$ enables the transition T_1. The valuations of the clocks are given by the matrix such that the initial valuation is $v_0 = \begin{matrix} \\ T_1 \end{matrix}\begin{matrix} \begin{matrix} \bullet & blue & red & black \end{matrix} \\ \begin{pmatrix} & & 0 & 0 \end{pmatrix} \end{matrix}$. Since the set of variables is $X = \{cpt\}$, we note a state $s = (m, cpt, v)$.

The initial state is $q_0 = (m_0, \{2,2,5\}, v_0)$. The transition T_1 is enabled twice and can fire after elapsing 8 time units for both enabling. After 8 time units T_1 fires with either the red or the black colors and then can fire again with the other one. Assume that we first fire with the red color, the corresponding run is as follows:

$$\left(m_0, \{2,2,5\}, \begin{matrix} \begin{matrix} \bullet & blue & red & black \end{matrix} \\ \begin{pmatrix} & & 0 & 0 \end{pmatrix} \end{matrix}\right) \xrightarrow{8} \left(m_0, \{2,2,5\}, \begin{matrix} \begin{matrix} \bullet & blue & red & black \end{matrix} \\ \begin{pmatrix} & & 8 & 8 \end{pmatrix} \end{matrix}\right)$$

$$\xrightarrow{(T_1,red)} \begin{pmatrix} & blue & red & \text{black} \\ P_1 & 1 & 0 & 1 \\ P_2 & 0 & 1 & 0 \\ P_3 & 1 & 0 & 0 \end{pmatrix}, \{2,4,5\}, \begin{pmatrix} \bullet & blue & red & black \\ & & & 8) \end{pmatrix}$$

$$\xrightarrow{(T_1,black)} \begin{pmatrix} & blue & red & \text{black} \\ P_1 & 1 & 0 & 0 \\ P_2 & 0 & 1 & 1 \\ P_3 & 2 & 0 & 0 \end{pmatrix}, \{2,4,4\}, \begin{pmatrix} \bullet & blue & red & black \\ & & &) \end{pmatrix}.$$

Example 2: The HCTPN given in Figure 3 illustrates time behavior and high-level manipulation of variables. This HCTPN has only one color and a single variable *cpt*, and is part of a larger HCTPN. We assume that *g()* returns an integer between 1 and 10, handled by the other part of the net.

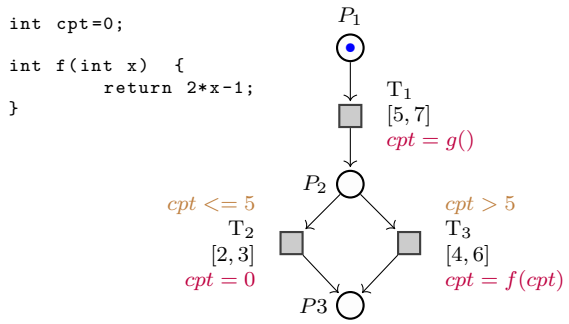

Figure 3. HCTPN illustrating high-level manipulation of variables.

A marking is written by the matrix (|*P*|,|*C*|). Since there is only one color, the marking is a vector and the initial marking is then $m_0 = \begin{matrix} P_1 \\ P_2 \\ P_3 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ and enables the transition T_1. The valuations of the clocks are given by the matrix (here a vector) such that the initial valuation is $v_0 = \begin{matrix} T_1 \\ T_2 \\ T_3 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. Since the set of variables is $X = \{cpt\}$, we note a state $s =$ (m,cpt,v). The initial state is $q_0 = (m_0,0,v_0)$. The transition T_1 can fire after elapsing 5 time units. We now consider the run where the function *g()* called by the update of the firing of T_1 returned the value 7. Then the transition's guard T_2 is false, and the transition T_3 is enabled. We assume that the transition T_3 took 4.6 time units for this run. The firing of the transition T_3 executes the corresponding update and calls the function *f* that returns 13. The corresponding run is as follows:

$$\begin{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, 0, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \end{pmatrix} \xrightarrow{5} \begin{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, 0, \begin{pmatrix} 5 \\ 0 \\ 0 \end{pmatrix} \end{pmatrix} \xrightarrow{(T_1,\bullet)} \begin{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, 7, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \end{pmatrix} \xrightarrow{4.6} \begin{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, 7, \begin{pmatrix} 0 \\ 0 \\ 4.6 \end{pmatrix} \end{pmatrix}$$

$$\xrightarrow{(T_3,\bullet)} \begin{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, 13, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \end{pmatrix}$$

Example 3: The model given in Figure 4 is a HCTPN with a set of two colors $C = \{red,blue\}$. Several combinations of color usage, on guards and in updates, via the $*any* variable are presented.

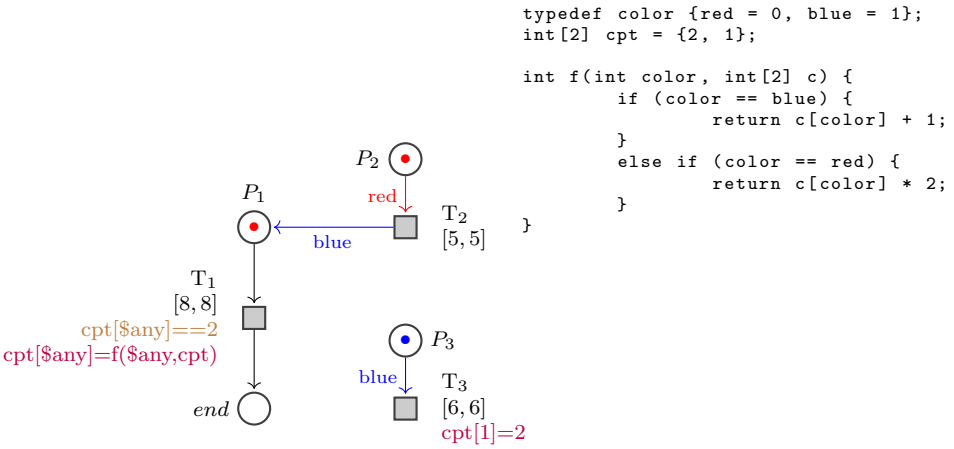

Figure 4. HCTPN model illustrating colored multi-enableness

In the sequel a marking is written by the matrix ($|P|,|C|$). The initial marking is then $m_0 = \begin{matrix} & red & blue \\ P_1 \\ P_2 \\ P_3 \\ end \end{matrix}\begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$ and enables the transitions T_1, T_2 and T_3. The variable *cpt* of the model is an array indexed by the color. Its initial value is $x_0 = cpt \begin{matrix} red & blue \end{matrix}\begin{pmatrix} 2 & 1 \end{pmatrix}$. The valuations of the clocks are given by the matrix such that the initial valuation is $v_0 = \begin{matrix} \bullet & red & blue \\ T_1 \\ T_2 \\ T_3 \end{matrix}\begin{pmatrix} 0 & 0 \\ 0 & \\ 0 & \end{pmatrix}$ (We omit the insignificant values). We note a state $s = (m,x,v)$. The initial state is $q_0 = (m_0,x_0,v_0)$. Since the time intervals are points, we have an unique run:

$$\left(\begin{pmatrix}1&0\\1&0\\0&1\\0&0\end{pmatrix},(2\quad 1),\begin{pmatrix}0&0\\0&\\0&\end{pmatrix}\right)\xrightarrow{5}\left(\begin{pmatrix}1&0\\1&0\\0&1\\0&0\end{pmatrix},(2\quad 1),\begin{pmatrix}5&0\\5&\\5&\end{pmatrix}\right)$$

$$\xrightarrow{(T_2,\bullet)}\left(\begin{pmatrix}1&1\\0&0\\0&1\\0&0\end{pmatrix},(2\quad 1),\begin{pmatrix}0&5&0\\0&&\\5&&\end{pmatrix}\right)\xrightarrow{1}\left(\begin{pmatrix}1&1\\0&0\\0&1\\0&0\end{pmatrix},(2\quad 1),\begin{pmatrix}0&6&0\\0&&\\6&&\end{pmatrix}\right)$$

$$\xrightarrow{(T_3,\bullet)}\left(\begin{pmatrix}1&1\\0&0\\0&0\\0&0\end{pmatrix},(2\quad 2),\begin{pmatrix}0&6&0\\0&&\\0&&\end{pmatrix}\right)\xrightarrow{2}\left(\begin{pmatrix}1&1\\0&0\\0&0\\0&0\end{pmatrix},(2\quad 2),\begin{pmatrix}0&8&2\\0&&\\0&&\end{pmatrix}\right)$$

$$\xrightarrow{(T_1,red)}\left(\begin{pmatrix}0&1\\0&0\\0&0\\1&0\end{pmatrix},(4\quad 2),\begin{pmatrix}0&0&2\\0&&\\0&&\end{pmatrix}\right)\xrightarrow{6}\left(\begin{pmatrix}0&1\\0&0\\0&0\\1&0\end{pmatrix},(4\quad 2),\begin{pmatrix}0&0&8\\0&&\\0&&\end{pmatrix}\right)$$

$$\xrightarrow{(T_1,blue)} \left(\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 1 \end{pmatrix}, (4 \quad 3), \begin{pmatrix} & 0 & 0 \\ 0 & & \\ 0 & & \end{pmatrix} \right)$$

The time elapses from the initial marking until reaching date 5. T_2 is fired, and a blue token is dropped in the place P_1. The clock of T_1 associated with the red color has reached the value 5. The clock of T_1 associated with the blue color cannot start yet because the guard is false for this color. At date 6, T_3 is fired, causing a change in the variable cpt that makes the guard of T_1 true for the blue color. The clock associated with the blue color for T_1 can therefore start. Both colors enable the transition T_1, and the corresponding clocks give the time from the two enabling. After two more time units, T_1 is fired for the red color; at this moment, the clock of T_1 for the blue color has reached 2. Finally, after 6-time units, T_1 is fired for the blue color, ending the run.

Atomicity: An update can be described as a sequence of imperative code expressed in a programming language such as C. This code is evaluated sequentially w.r.t. the semantics of the C language; however, its execution is considered atomic from the HCTPN point of view.

Hence, if x and x' are respectively the values of the variables before and after the execution of the code of an update of a transition t from x, the firing of t leads atomically to $x' = update(t,x)$.

High-level Colored Time Petri Net with stopwatches

We now consider stopwatches instead of clocks. Hence, for Colored Time Petri Nets with stopwatches, there is at most one stopwatch per color and per transition.

When using stopwatches with the HCTPN formalism, the temporal behavior differs and depends on the time derivative function $\dot{v}(t,\bullet)$ when transitions $t \in T_{\overline{any}}$ and $\dot{v}(t,c)$ for $t \in T_{any}$.

The time associated with a transition can be suspended and later resumed at the same point. Moreover, transition with a 0-time derivative cannot be fired. The time derivative of a stopwatch is in the rate set $\{0,1\}$ and is given by a function from Markings.

Definition 3 (High-level Colored Time Petri Net with stopwatches).

A High-level Colored Time Petri Net with stopwatches is a tuple $\mathcal{N} = (P,T,\mathsf{pre}(.),\mathsf{post}(.),m_0,guard,update,I,\dot{v})$ where $(P,T,\mathsf{pre}(.),\mathsf{post}(.),m_0,guard,update,I)$ *is* defined in Definition 1 and $\dot{v} : T \times \mathbb{N}^{P\times C} \times \mathbb{X}^X \rightarrow \{0,1\}$ is the time derivative function.

Semantics

For the discrete transition of the semantics, the only difference with HCTPN is that a transition cannot be fired if its time derivative is not 1. For the time transition, the value of a stopwatch evolves according to its derivative as follows.

Definition 4 (Semantics of a HCTPN with stopwatches). The semantics of a HCTPN with stopwatches is a timed transition system $(Q, Q_0, \rightarrow)$ where:

- $Q \subseteq \mathbb{N}^{P\times C} \times \mathbb{X}^X \times {\mathbb{R}_{\geq 0}}^{T\times C}$

- $Q_0 = ((m_{0,}x_0), \bar{0})$
- $\rightarrow \in Q \times ([T \times C \cup \{\bullet\}) \cup \mathbb{R}_{\geq 0}) \times Q$ consists of two types of transitions:
    - discrete transitions (firing *t* from ((*m*,*x*),*v*)), as presented in Definition 2 with $\dot{v}(t) = 1$
    - time transitions: $((m, x), v) \xrightarrow{d \in \mathbb{R}_{\geq 0}} ((m, x), v')$, iff
        - $\forall t \in T_{\overline{any}}$ s.t. $en(m, t) = true$,
            - $v'(t,\bullet) \leq I(t) \downarrow$
            - $v'(t,\bullet) = v(t,\bullet) + d$ if $\dot{v}(t,\bullet) = 1$ otherwise $v'(t,\bullet) = v(t,\bullet)$
        - $\forall t \in T_{\overline{any}}$ s.t. $en(m, t) = false$,
            - $v'(t,\bullet) = 0$
        - $\forall t \in T_{any}$ and $\forall c \in \text{colorSet}_{\text{any}}(m,t)$,
            - $v'(t, c) \leq I(t) \downarrow$
            - $v'(t, c) = v(t, c) + d$ if $\dot{v}(t, c) = 1$ otherwise $v'(t, c) = v(t, c)$
        - $\forall t \in T_{any}$ and $\forall c \notin \text{colorSet}_{\text{any}}(m,t)$,
            - $v'(t, c) = 0$

We now illustrate the main features of HCTPN with stopwatches on an example.

Example of HCTPN with stopwatches

This example is the modeling of the preemptive scheduling of two tasks. The first task $task_1$ is a periodic task running on core 0, assigned to blue color. The second task $task_2$ is also periodic but is executed only 10 times on core 1, assigned to red color. The particular color *any* is used for enabling and firing all transitions. For the first two executions of $task_2$, the priority of $task_1$ is higher than $task_2$ priority, after which it becomes the opposite.

The model in Figure 5 is a HCTPN with stopwatches and has a single shared variable cpt and two colors. The initial value of cpt is zero. Only the transition T_2 has a guard and an update that manipulate the cpt variable. Hence the transition T_2 is enabled if there is a token in its input place $task_2$ and if $cpt < 10$ modeling the fact that the task $task_2$ is executed only 10 times. The update that increments the value of cpt is executed each time the transition T_2 is fired.

The scheduling is captured by the derivative function of the stopwatches associated with C_1 and C_2 whose values are given by a function called *isRunning* shown in Figure 5.

In the sequel, a marking is written by the matrix $m = (|P|,|C|)$. The initial marking enables the transitions T_1 and T_2. The valuations of the stopwatches are given by the matrix $v = (|T|,|C|)$. Since all the transitions are in T_{any}, the bullet column of the stopwatch valuations is not used. We will therefore omit it in the states of this example in order to simplify the notation.

```
typedef enum {task1,task2} id;
int cpt = 0;

int isRunning(id task)  {
  if (task==task1) {
    if ((m(Ready2)==1) && (cpt>2)) return 0; else return 1;
  } else if (task==task2) {
    if ((m(Ready1)==1) && (cpt<3)) return 0; else return 1;
  }
}
```

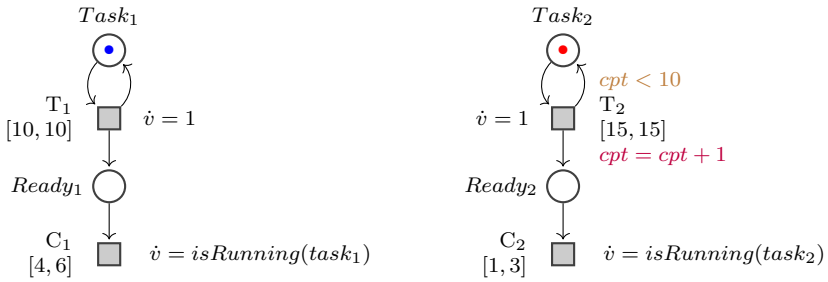

Figure 5. HCTPN model with stopwatches of two-task scheduling.

We note a state $s = (m,cpt,v)$ and the initial state is $q_0 = (m_0,0,v_0)$, where:

$$m_0 = \begin{matrix} \\ Task_1 \\ Task_2 \\ Ready_1 \\ Ready_2 \end{matrix} \begin{matrix} red & blue \end{matrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \text{ and } v_0 = \begin{matrix} \bullet \\ T_1 \\ T_2 \\ C_1 \\ C_2 \end{matrix} \begin{matrix} red & blue \end{matrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \text{ simply denoted } v_0 = \begin{matrix} \\ T_1 \\ T_2 \\ C_1 \\ C_2 \end{matrix} \begin{matrix} red & blue \end{matrix} \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Assume that the execution times of the two tasks $task_1$ and $task_2$ are respectively 5.3 and 2.4. It means that the transitions C_1 and C_2 fire when their stopwatches reach these values. Let us develop the corresponding run:

$$q_0 = \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, 0, \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \right) \xrightarrow{10} \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, 0, \begin{pmatrix} 0 & 10 \\ 10 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \right) \xrightarrow{(T_1,blue)} q_1$$

$$= \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}, 0, \begin{pmatrix} 0 & 0 \\ 10 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \right)$$

In q_1, we have $\dot{v}(C_1, blue) = 1$ then

$$q_1 \xrightarrow{5} q_2 = \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}, 0, \begin{pmatrix} 0 & 5 \\ 15 & 0 \\ 0 & 5 \\ 0 & 0 \end{pmatrix} \right) \xrightarrow{(T_2,red)} q_3 = \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}, 1, \begin{pmatrix} 0 & 5 \\ 0 & 0 \\ 0 & 5 \\ 0 & 0 \end{pmatrix} \right)$$

In q_3, we have $\dot{v}(C_1, blue) = 1$ and $\dot{v}(C_2, red) = 0$ meaning that $task_2$ is preempted by $task_1$. Then $v(C_2, red)$ will keep its value 0 until the firing of C_1 that will change

$\dot{v}(C_2, red)$.

$$q_3 \xrightarrow{0.3} q_4 = \left(\begin{pmatrix}0&1\\1&0\\0&1\\1&0\end{pmatrix}, 1, \begin{pmatrix}0&5.3\\0.3&0\\0&5.3\\0&0\end{pmatrix}\right) \xrightarrow{(C_1,blue)} q_5 = \left(\begin{pmatrix}0&1\\1&0\\0&0\\1&0\end{pmatrix}, 1, \begin{pmatrix}0&5.3\\0.3&0\\0&0\\0&0\end{pmatrix}\right)$$

In q_5, we have $\dot{v}(C_2, red) = 1$ hence

$$q_5 \xrightarrow{2.4} q_6 = \left(\begin{pmatrix}0&1\\1&0\\0&0\\1&0\end{pmatrix}, 1, \begin{pmatrix}0&7.7\\2.7&0\\0&0\\2.4&0\end{pmatrix}\right) \xrightarrow{(C_2,red)} q_7 = \left(\begin{pmatrix}0&1\\1&0\\0&0\\0&0\end{pmatrix}, 1, \begin{pmatrix}0&7.7\\2.7&0\\0&0\\0&0\end{pmatrix}\right)$$

For the sake of conciseness, we do not detail the following run from q_7

$$q_7 \xrightarrow{2.3} \xrightarrow{(T_1,blue)} \xrightarrow{6} \xrightarrow{(C_1,blue)} \xrightarrow{4} \xrightarrow{(T_1,blue)} \xrightarrow{(T_2,red)} \xrightarrow{6} \xrightarrow{(C_1,blue)} \xrightarrow{3} \xrightarrow{(C_2,red)} \xrightarrow{1} \xrightarrow{(T_1,blue)} \xrightarrow{5} \xrightarrow{(T_2,red)} q_{13}$$

It leads to a state q_{13} that have exactly the same marking and the same value of stopwatches than q_3 but with *cpt* = 3.

$q_{13} = \left(\begin{pmatrix}0&1\\1&0\\0&1\\1&0\end{pmatrix}, 3, \begin{pmatrix}0&5\\0&0\\0&5\\0&0\end{pmatrix}\right)$ then we have $\dot{v}(C_1, blue) = 0$ and $\dot{v}(C_2, red) = 1$

meaning that the task *task*$_2$ is not preempted by the task *task*$_1$. Hence, we have:

$$q_{13} \xrightarrow{2.4} q_{14} = \left(\begin{pmatrix}0&1\\1&0\\0&1\\1&0\end{pmatrix}, 3, \begin{pmatrix}0&7.4\\2.4&0\\0&5\\2.4&0\end{pmatrix}\right) \xrightarrow{(C_2,red)} q_{15} = \left(\begin{pmatrix}0&1\\1&0\\0&1\\0&0\end{pmatrix}, 3, \begin{pmatrix}0&7.4\\2.4&0\\0&5\\0&0\end{pmatrix}\right)$$

Decidability, complexity and state space abstraction

Let us recall that a *High-level Colored Time Petri Net* (HCTPN) is a tuple $\mathcal{N}$ = (*P,T,X,C,*pre,post,(m_0,x_0),*guard,update,I*) such that the set *C* of *colors* is finite and *X* is a finite set of *variables* taking their value in a finite set $\mathbb{X}$.

A *state* q of the net $\mathcal{N}$ is a tuple $((m, x), v)$ in $\mathbb{N}^{P\times C} \times \mathbb{X}^X \times {\mathbb{R}_{\geq 0}}^{T\times C}$, where: m is a colored marking, x is a variable valuation, (m, x) is the discrete part of the state and v is a valuation of the clocks. The semantics of $\mathcal{N}$ is (Q,Q_0,→).

We propose to extend the zone based graph of (Boucheneb, Gardey, and Roux (2009)) to HCTPN. Note that we can similarly extend the state class graph of (Berthomieu and Diaz (1991)).

Symbolic state. For an arbitrary sequence of discrete transitions σ= (t_i,c_j)…(t_m,c_n), let C_σ be the set of all states that can be reached by the sequence σ from q_0: $C_\sigma = \{q \in Q | q_0 \xrightarrow{(t_i,c_j)} q_1 \ldots \xrightarrow{(t_m,c_n)} q\}$. All the states of C_σ share the same marking *m*, the same variable values *x* and can therefore be written as a pair ((*m*,*x*),*Z*) where (*m*,*x*) is the common discrete state and *Z* is the union of the points (valuation of the clocks) allowing this sequence. *Z* is called a zone and can be written as a set of linear inequations. Hence C_σ is called a symbolic state.

We denote ≡, the relation between two zones when they represent the same set of values (independently of their syntactical representation). For Z and Z', two systems of linear inequations over a set of variables Δ, we denote $Z \equiv Z'$ when they have equal solution sets over Δ.

We denote ≅, the relation satisfied by two such sets of states when they have the same discrete part and the same firing domain.

Definition 5. Let $C_\sigma = ((m, x), Z)$ and $C_{\sigma'} = ((m', x'), Z')$ be two sets of states, $C_\sigma \cong C_{\sigma'}$ iff $m = m', x = x' \; and \; Z \equiv Z'$.

In HCTPN, a transition can be multi-enabled a maximum of $|C|$ times at a given time. Hence D is defined over $|C|\times|T|$ variables. If $C_\sigma \cong C_{\sigma'}$, any firing schedule firable from some state in C_σ is firable from a state in $C_{\sigma'}$ and conversely. As for classical Time Petri Nets, the symbolic states are the above sets C_σ considered modulo ≅ equivalence.

Definition 6. The zone based graph (ZBG) is defined by the set of symbolic states equipped with a transition relation: $C_\sigma \xrightarrow{t} C'$ iff $C_{\sigma.t} \cong C'$. Hence the ZBG computes the smallest set C of symbolic states w.r.t. ≅.

Theorem 5.1. *The ZBG of an HCTPN is finite iff the net is bounded. Moreover, the ZBG is a complete and sound state space abstraction of the HCTPN.*

Proof. Let $n = |C| + 1$ and $m = |T|$. Given a symbolic state $C = ((m, x), Z)$, a point $\delta = \left(\delta_{(1,1)}, \dots, \delta_{(1,n)}, \delta_{(2,1)}, \dots, \delta_{(m,n)}\right) \in Z$ where $\delta_{(i,j)}$ is the clock that refers to $v(t_i, c_j)$ with $t_i \in T$ and $c_j \in C \cup \{\bullet\}$. The zone may be described by linear inequations of the form $\delta_{(i,j)} \leq k$ or $\delta_{(i,j)} - \delta_{(i,j)} \leq k'$ where $k \in \mathbb{N}$ and $k' \in \mathbb{Z}$. Hence, for HCTPN, as in (Boucheneb, Gardey, and Roux (2009)), zones can be symbolically abstracted using so-called Difference Bound Matrix (DBM) over $n \times m$ variables, and the number of DBM is finite[2]. Moreover, by assumption, the number of markings is bounded, and by definition, $\mathbb{X}$ is a finite set, then the number of discrete states $((m, x))$ is also bounded. Hence the number of nodes of the SCG is bounded. The completeness and soundness of the ZBG directly come from (Boucheneb, Gardey, and Roux (2009)).

Corollary 5.2. Reachability problem for bounded High-level Colored Time Petri Net is decidable.

Proof. From theorem 5.1, zones can be symbolically abstracted using Difference Bound Matrix (DBM) which are computable finite abstractions leading to the ZBG. The ZBG preserves untimed language and reachability (Boucheneb, Gardey, and Roux, 2009) (as state class graph (Berthomieu and Diaz, 1991)), then the reachability problem is decidable.

Implementation: The zone based graph as the state class graph algorithms for HCTPN are implemented in ROMÉO (Lime et al. (2009)). Temporal logics were introduced by Pnueli (Pnueli (1977)) as specification languages to express the behaviors of sequential and concurrent systems, and TCTL (Timed Computation Tree Logic), introduced in (Alur, Courcoubetis, and Dill (1993)), is a real-time extension of the branching-time temporal logic CTL (Computation Tree Logic).

---

[2] For static firing interval with infinite time upper bound, the abstraction may use a k-approximation operator on zones to enforce the termination where k is the largest constant of the model.

We can prove, as in (Boucheneb, Gardey, and Roux (2009)) for bounded Time Petri Nets, that the theoretical complexity of TCTL model-checking (and then reachability problem) for bounded High-level Colored Time Petri Nets is PSPACE-complete. However, as for Timed Automata and Time Petri Nets, no effective PSPACE algorithm exists in practice, and real implementations are with exponential algorithms.

In practice, on-the-fly TCTL model-checking for bounded High-level Colored Time Petri Nets is proposed in the ROMÉO tool,

Stopwatch setting: As shown in the previous section, HCTPN can be extended with stopwatches allowing the modeling of preemptive scheduling. Let us now consider HCTPN with stopwatches.

Theorem 5.3. Reachability problem for bounded High-level Colored Time Petri Net with stopwatches is undecidable.

Proof. Any stopwatch Petri Net (Berthomieu et al., 2007) can be trivially encoded by HCTPN with stopwatches with only one color. Reachability is undecidable for Stopwatch Petri Net (Berthomieu et al., 2007). □

In the stopwatch setting, the reachability problem is undecidable. The symbolic abstraction given in definition 6 with the zone based graph remains correct but can no longer be encoded by a DBM. The finiteness of the ZBG is no longer guaranteed, but the linear inequation set of a zone can be encoded by convex polyhedra, and the ZBG can be obtained (when finite) by a semi-algorithm.

For HCTPN with stopwatches, polyhedral semi-algorithm extensions of the zone based graph and the state class graph are implemented in ROMÉO (Lime et al. (2009)) and converge for almost all practical cases.

Path finder monocore scheduling example

We propose to model the well-known pathfinder scheduling problem (Jones (1997)). Pathfinder's Sojourner Rover rolled onto Mars' surface on July 6, 1997. It used IBM's RS6000 processor and Wind River vxWorks RTOS. VxWorks provides preemptive fixed-priority scheduling and tasks were executed as threads with priorities determined by their relative urgency. The system has a number of periodic tasks of varying priority such as the low priority. Atmospeheric Structure Instrument and Meteorology Task (ASI/MET). The period of the ASI/MET task is 5s and its execution time is dependent on the weather conditions and is therefore in a time interval between 50 and 75ms. The data from the lander must be transmitted via the 1553 bus to the radio in order to be transmitted to Earth. Moreover, the control signal from the CPU must be transmitted via the 1553 bus to the cruise part and the lander part. The management of 1553 bus is implemented as two critical tasks: bc_sched (bus scheduler task) and bc_dist (bus distribution task).

The data structure of the pipe (memory) associated to the transmitted data is a shared resource protected by mutex. Hence the bc_dist (a higher priority task) and ASI/MET (a lower priority task) both share the data structure of the pipe. We consider seven periodic tasks summarized in the following table:

Table 1. Periodic tasks.

Task	Priority	Execution Time (ms)	Period (ms)	Shared ressource access
bc_sched	7	25	125	

bc_dist	6	25	125	yes
Control	5	25	250	yes
Radio	4	25	250	
Camera	3	25	250	
Measurement	2	50	5000	yes
ASI/MET	1	[50,75]	5000	yes

As the tasks are not independent (shared resources) and as the execution times are in intervals, the classical analytical schedulability methods are difficult to apply. We model the path finder task configuration in HCTPN (with one color since there is only one core) and use the Roméo tool to check the schedulability. All the tasks are modelled as the ASI/MET task shown in Figure 6. Tasks that do not access the shared resource have the same model without the arcs linking to the Shared Ressource place. The isRunning function is the classic scheduling function: it returns 1 if the task is the highest priority ready task and 0 otherwise. A task waiting for the non available resource is not considered ready.

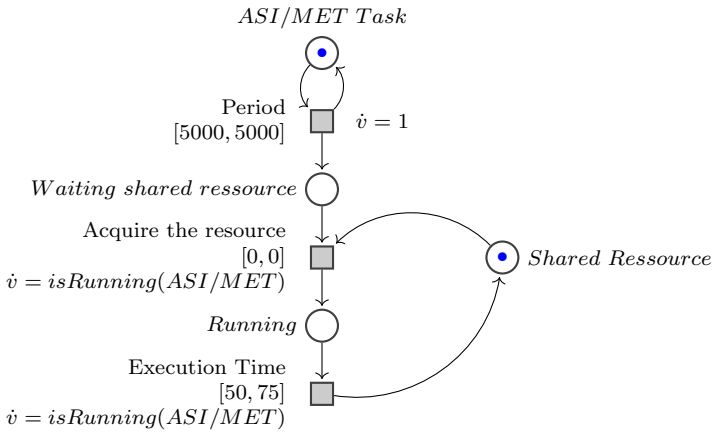

Figure 6: HCTPN model of ASI/MET task.

The schedulability property is that for each task the sum of the tokens in the places *Waiting shared ressource* and *Running* must be at most 1. It is checked by the CTL formula AG(*Waiting shared ressource* + *Running* ≤ 1). The model checker replies that this is not true and gives a counter example which shows that the *ASI/MET* task can acquire the resource before the *bc_dist* task wakes up. The latter then takes the processor and blocks while waiting for the resource. The *ASI/MET* task then resumes its execution but is preempted by the *Radio* and *Camera* tasks with a higher priority than it but with a lower priority than the *bc_dist* task, resulting in a deadline overrun. The problem appears when the *ASI/MET* task has an execution time close to its maximum bound and is therefore a difficult phenomenon to observe in practice.

Indeed, a few days after landing, the spacecraft began to undergo total system resets, each time resulting in data loss. The source of the problem was due to a priority inversion which caused a deadline-miss of a critical task (*bc_dist*), resulting in a spacecraft reset.

Application

The application chosen as an example is the modeling of the spinlocks mechanism present in the PowerPC MPC5643L dual-core microcontroller from NXP (Freescale Semiconductor (2013)) and used to build critical sections for parallel program executions. This mechanism is based on a hardware unit, the SEMA4 for *Semaphore Unit*. For the

software, this unit is materialized as an array of 16 registers implementing 16 locks. The exclusive access to the bus regulates the concurrent accesses to one of these registers. If a register contains the value 0, the lock is available, and it is possible to write to it. If the value contained is different from 0, the lock is occupied, and it is only possible to write the value 0 to it, and writing any other value has no effect. Therefore, getting a lock consists in writing a value different from 0 and releasing it consists in writing 0. Thus, using this unit requires respect of a protocol, and an example of implementation is given on page 1322 of (Freescale Semiconductor (2013)). A simpler version is given by the algorithm 1.

Algorithm 1 Lock acquisition protocol. gate is one of the hardware registers of the SEMA4 unit.

```
cn ← core_number     ▷ (1 ... N)
do
        gate ← cn
        lock ← gate
while lock ≠ cn
```

Here, it is assumed that the waiting execution thread loops in active wait until acquiring the lock. The SEMA4 unit also proposes to notify the release of the lock by means of an interrupt. This allows the mechanism to be coupled to an operating system and to schedule tasks according to whether the lock is taken or not. For this example, the operating system offers the following features:

- The scheduler has a fixed priority;
- The tasks can be in the following states:
  (1) SUSPENDED, the task is not started;
  (2) READY, the task is ready to be executed and eligible by the scheduler;
  (3) RUNNING, the task is currently executing;
  (4) WAITING, the task is waiting.
- The following services are available:
  (5) Activate() moves a task from SUSPENDED to READY;
  (6) Terminate() moves a task from RUNNING to SUSPENDED;
  (7) Wait() moves a task from RUNNING to WAITING;
  (8) UnWait() moves a task WAITING to RUNNING;
  All these services cause a rescheduling. These features are illustrated in the Figure 7.

The combination of the SEMA4 unit with the operating system leads to defining a second algorithm for the lock acquisition protocol as shown in listing 2. In essence, it is a matter of calling the Wait service if the lock is not available, which will allow to schedule another task instead of doing an active wait. Moreover, the release of the lock leads to the sending of an interrupt and the interrupt handler has the role of calling UnWait() to put the task in the READY state.

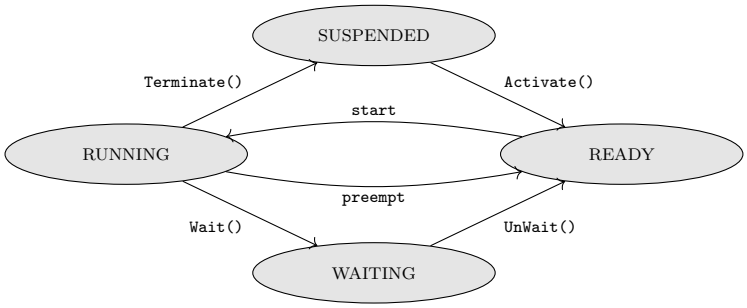

Figure 7: Possible states for a task and interaction with services. start and preempt are performed by the scheduler according to the changes of state of the other tasks.

Algorithm 2. Lock acquisition protocol in combination with the operating system.

gate is one of the hardware registers of the SEMA4 unit.

```
cn ← core_number      ▷ (1 ... N)
do
    gate ← cn
    lock ← gate
    if lock ≠ cn then
      Wait()
    end if
while lock ≠ cn
```

Modeling the operating system

We consider that the operating system runs only on core 0 and that core 1 executes a single task program. The state of a task includes its priority and its state (SUSPENDED, READY, RUNNING or WAITING) and is represented by a structure. The state of the operating system is also represented by a structure: kern. It includes:

- A table of tasks;
- The id of the task currently running;
- The id of the task in the WAITING state;
- If an interrupt is pending;
- A lock to have a critical section in the operating system.

The listing 1 presents these structures.

Listing 1 Data structures used for operating system status

```
 1 typedef struct {
 2   int prio;
 3   int state;
 4 } TaskDescriptor;
 5
 6 typedef struct {
 7   TaskDescriptor[TASK_COUNT] tasks;
 8   int running;
 9   int waiting;
10   int it_pending;
11   int locked;
12 } KernelState;
```

The services of the operating system are also modeled by functions that will be called in the updates on the transitions. As an example, the listing 2 presents the service Wait. After having verified that the caller is indeed the task being executed (line 3), the service preempts the task: it is removed from the running field (line 4), its state is changed to WAITING (line 5), it is placed in the waiting field (line 6) and then, finally, a rescheduling is performed (line 7).

Listing 2 Wait service

```
1 void Wait(int inCaller)
2 {
3   if (inCaller == kern.running) {
4     kern.running = NONE;
5     kern.tasks[inCaller].state = WAITING;
6     kern.waiting = inCaller;
7     Schedule();
8   }
9 }
```

Modeling the spinlocks mechanism

The modeling takes advantage of the possibilities of the HCTPN. The hardware part, which by virtue of the exclusive access to the bus allows operations that are intrinsically atomic, is modeled using functions. To simplify the presentation, only one register of the SEMA4 unit, *g*, is modeled but the model could just as well use an array to accurately model the hardware. The listing 3 shows this part of the model. *g* is defined as a structure comprising two members: (1) *gate* which is the state variable and (2) *it* which is used to memorize the fact that the release should cause an interrupt to be sent. *gate* is initialized to the UNLOCKED state, and *it* is initialized to NO_IT (line 10). *g* is accessible through three functions. *lock* (line 12) mimics the behavior of the hardware by only allowing writing to *gate* if its value is UNLOCKED. The core number corresponding to a color and a color among N being coded by an integer from 0 to N-1, a core locks by writing *color* + 1 in *gate*. *unlock* (line 18) simply writes the value UNLOCKED into *gate*. Finally, *isLockedBy* (line 22) returns 1 if *core* holds the lock and returns 0 otherwise.

Each function of the software and each task is modeled by an HCTPN with stopwatches reproducing the control flow graph of the function and allowing to preempt the execution as shown in 4.2. Three HCTPNs model the functions *GetSpinLock*, *GetSpinLockIT* and *RelSpinLock* (see Figure 8. *GetSpinLock* corresponds to the algorithm 1 and *$any* allows to represent on which core the function is executed. *GetSpinLockIT* corresponds to the algorithm 2. The call of a function modeled by a HCTPN is done by dropping a token of the core color in the initial place. Thus, “calling” the function *GetSpinLock* is performed by the update GetSpinLock[*color*] = 1 on a transition of the HCTPN of the calling function. This is identical to drawing an arc of the corresponding color between the transition and the initial place of *GetSpinLock*. The function return requires a synchronization. This one is implemented by a variable of type array and of size equal to the number of colors and indexed by the color, i.e. the core on which the function call is made. We have therefore for our three function models the three variables endOfGSL, endOfGSLIT and endOfRSL, see listing 4. The locked field of kern is used to have a critical section between *GetSpinLockIT* and *RelSpinLock* and to prevent a situation where the spinlock would be released while in *GetSpinLockIT* the token would be in P_3 with the result that the interrupt would not be triggered. Finally, all transitions include a derivative function of the stopwatches: $\dot{v} = isRunning(\$any,caller)$. This function returns 1 if the calling task (*caller* is a variable that is assigned with the identifier of the calling task) is running and 0 otherwise.

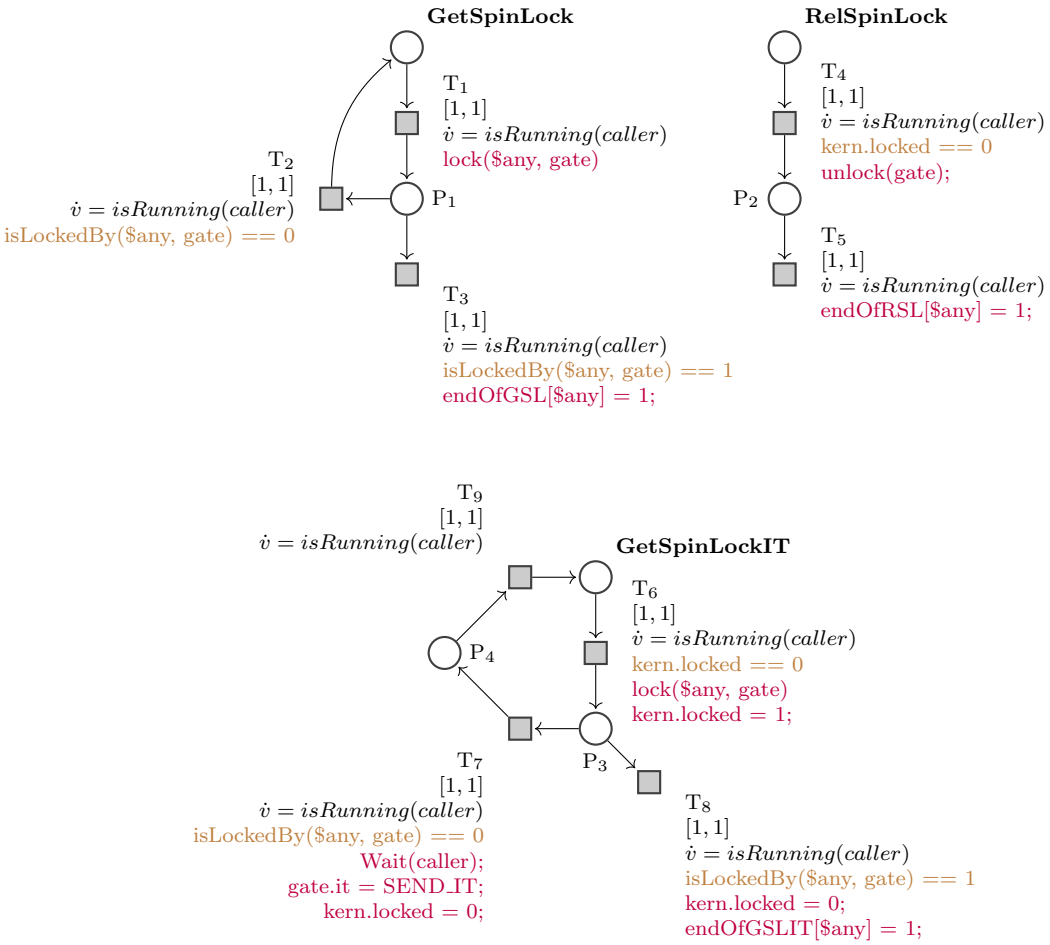

Figure 8. The GetSpinLock, GetSpinLockIT and RelSpinLock function models

Listing 3 Modeling of the SEMA4 hardware

```
typedef struct {
  int gate;
  int it;
} SEMA4Gate;

const int UNLOCKED = 0;
const int NO_IT = 0;
const int SEND_IT = 1;

SEMA4Gate gate = { UNLOCKED, NO_IT };

void lock(int core, int &ioGate) {
    if (ioGate == UNLOCKED) {
        ioGate = core + 1;
    }
}

void unlock(int &ioGate) {
    ioGate = UNLOCKED;
}

int isLockedBy(int core, int &inGate) {
    if (inGate == core + 1) {
        return 1;
    } else {
        return 0;
    }
}
```

Listing 4 Synchronization variables for the function return

```
1 int[2] endOfGSL   = {0, 0};
2 int[2] endOfGSLIT = {0, 0};
3 int[2] endOfRSL   = {0, 0};
```

In *RelSpinLock*, calling the unlock function while a task is waiting for the spinlock triggers an interrupt by setting the it_pending field of the kern structure to 1. The interrupt handler, which runs on core 0, is then triggered. It acknowledges the interruption by resetting it_pending and calls UnWait() (see figure 9). As a result, the task τ_0 is scheduled instead of τ_1.

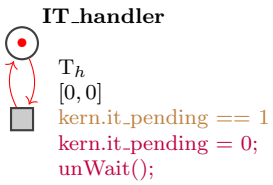

Figure 9. The interrupt handler which is triggered when the spinlock is released with a task waiting for it.

Verification of the system

The spinlock model and operating system model are completed by an application model. Two tasks, τ_0 and τ_1, running on core 0 (red color) and τ_2, running on core 1 (blue color), are modeled as shown in Figure 10. The task τ_2, being alone on core 1, cannot be preempted. It is assigned the identifier -1 and *isRunning* always returns 1 for a task with identifier -1. The task τ_0 takes then releases the spinlock. Task τ_2 has the possibility to take it, as τ_0 does, or to reach the final state without taking the spinlock. If τ_2 owns the spinlock when τ_0 tries to take it, τ_0 goes into the WAITING state and the operating system schedules τ_1 instead.

We want to check the following properties:

- τ_0 and τ_2 cannot occupy simultaneously and respectively the places P_{12} and P_{22}. It is checked by the CTL formula $A\Box(\neg(P_{12}[0] == 1 \wedge P_{22}[1] == 1))$. Here $P_{12}[0]$ denotes the marking of P_{12} for the red color and $P_{22}[1]$ denotes the marking of P_{22} for the blue color;
- τ_0, τ_1 and τ_2 all reach their final places. That is to say that the use of the spinlock did not cause any blocking and that the scheduling done by the operating system allowed τ_0 and τ_1 to run until the end. It is checked by $A\Diamond(P_{14}[0] == 1 \wedge P_{24}[1] == 1 \wedge P_{34}[0] == 1)$;
- it happens τ_0 reaches its final place while τ_1 does not reach its final place yet. This is checked by the CTL formula $E\Diamond(P_{14}[0] == 1 \wedge P_{34}[0] == 0)$.

ROMÉO answers *true* for all these three CTL formula.

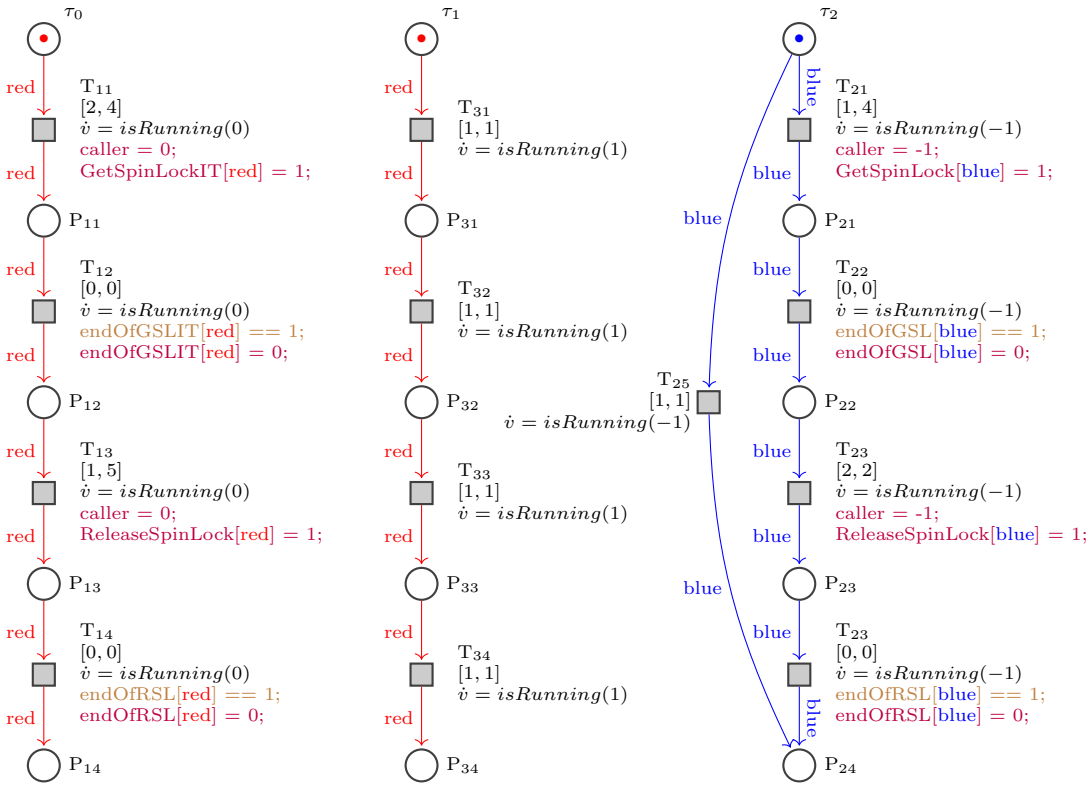

Figure 10. The tasks models

Conclusion

This paper has presented High-level Colored Time Petri Nets with stopwatches. This formalism allows to model multi-core complex systems allowing to preempt the execution, as shown in the case study. The high-level features allow the modeling of the software, the timed transitions model the execution times, the colors specify the hardware where the software is executed, and preemption is supported by means of stopwatches. A timed transition enabled by more than one color allows true concurrency modeling. The model-checking of this extended formalism is implemented in the ROMÉO tool. Future work will focus on using this formalism for the certification of an OSEK and AUTOSAR compliant embedded OS.

References

Alur, R., C. Courcoubetis, and D. Dill. 1993. "Model-Checking in Dense Real-time." Information and Computation 104 (1): 2-34.

Alur, Rajeev, and David L. Dill. 1994. "A theory of timed automata." Theoretical Computer Science.

Berthomieu, B., and M. Diaz. 1991. "Modeling and Verification of Time Dependent Systems Using Time Petri Nets." IEEE Trans. on Soft. Eng. 17 (3): 259-273.

Berthomieu, Bernard, Didier Lime, Olivier H. Roux, and Francois Vernadat. 2007. "Reachability Problems and Abstract State Spaces for Time Petri Nets with Stopwatches." Journal of Discrete Event Dynamic Systems - Theory and Applications (DEDS) 17 (2): 133-158.

Boucheneb, Hanifa, Guillaume Gardey, and Olivier H. Roux. 2009. "TCTL model checking of Time Petri Nets." Journal of Logic and Computation 19 (6): 1509-1540.

Boyer, Marc, and Michel Diaz. 2001. "Multiple Enabledness of Transitions in Petri Nets with Time." In Proceedings of the 9th International Workshop on Petri Nets and Performance Models, PNPM 2001, Aachen, Germany, September 11-14, 2001, 219-228. IEEE Computer Society.

Boyer, Marc, and Olivier H. Roux. 2008. "On the compared expressiveness of arc, place and transition time Petri Nets." Fundamenta Informaticae 88 (3): 225-249.

Bucci, G., A. Fedeli, L. Sassoli, and E. Vicario. 2004. "Time state space analysis of real-time preemptive systems." IEEE transactions on software engineering.

Cassez, Franck, and Kim Larsen. 2000. "The Impressive Power of Stopwatches." In CONCUR 2000 - Concurrency Theory, .

Clarke, Edmund M., Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. 2018. Hand- book of Model Checking. Springer Publishing Company, Incorporated.

Freescale Semiconductor. 2013. MPC5643L Microcontroller Reference Manual. NXP.

Haur, Imane, Jean-Luc Bechennec, and Olivier H. Roux. 2022. "High-level Colored Time Petri Nets for true concurrency modeling in real-time software." In International Conference on Control, Decision and Information Technologies (CODIT 2022), Istanbul, Turkey, May.

Henzinger, T.A., X. Nicollin, J. Sifakis, and S. Yovine. 1994. "Symbolic Model Checking for Real-Time Systems." Information and Computation.

Hillah, Lom, F. Kordon, Laure Petrucci, and Nicolas Treves. 2006. "PN Standardisation: A Survey." In Formal Techniques for Networked and Distributed Systems - FORTE 2006, Vol. 4229 of Lecture Notes in Computer Science, 307-322. Springer.

Jones, Mark. 1997. "What really happened on mars rover pathfinder." .

Kindler, Ekkart, and Laure Petrucci. 2009. "A framework for the definition of variants of high- level Petri nets." In Proceedings of the Tenth Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools (CPN '09), 121-137.

Lime, Didier, and Olivier H. Roux. 2003. "Expressiveness and analysis of scheduling extended time Petri nets." IFAC Proceedings Volumes .

Lime, Didier, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. 2009. "Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches." In 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009), edited by Stefan Kowalewski and Anna Philippou, Vol. 5505 of Lecture Notes in Computer Science, York, United Kingdom, Mar., 54-57. Springer.

Merlin, P., and D. Farber. 1976. "Recoverability of Communication Protocols - Implications of a Theoretical Study." IEEE Transactions on Communications.

Pnueli, Amir. 1977. "The Temporal Logic of Programs." In 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 46-57. IEEE Computer Society. Ramchandani, C. 1974. Analysis of asynchronous concurrent systems by timed petri nets. Technical Report.

Roux, Olivier, and Anne-Marie Deplanche. 2002. "A T-time Petri net extension for real time-task scheduling modeling." European Journal of Automation.

Roux, Olivier H., and Didier Lime. 2004. "Time Petri Nets with Inhibitor Hyperarcs. Formal Semantics and State Space Computation." In Applications and Theory of Petri Nets 2004,.