Design and verification of pipelined circuits with Timed Petri Nets[*]

Rémi Parrot[1], Mikaël Briday[1] and Olivier H. Roux[1]

[1]Nantes Université, École Centrale de Nantes, LS2N UMR CNRS 6004, 1 rue de la Noë, Nantes, 44300, France.

Contributing authors: remi.parrot@ec-nantes.fr; mikael.briday@ec-nantes.fr; olivier-h.roux@ec-nantes.fr;

Abstract

A fundamental step in circuit design is the placement of pipeline stages, which can drastically increase the data throughput. *Retiming* allows optimizing the pipeline with regard to a criterion, for example the required number of registers. This article presents an extension of Timed Petri Net to model synchronous electronic circuits, in order to explore the design space of pipelines. The Timed Petri Nets "à la Ramchandani" with a maximal step firing rule, have notably been used for the modeling of electronic circuits. The RTPN extension, through the *reset* which model the pipeline stages, and through the *delayable* transitions which relax some temporal constraints, makes it possible to widen the design space of pipelined systems, and thus to deal with both the retiming and the verification. After a formal definition of this model, we present a method to explore pipelines verifying temporal properties. We apply our approach to a time-multiplexing property allowing the mutualization of operators while minimizing the number of registers.

Keywords: pipeline optimization, model checking, Timed Petri Net, resource sharing, time-multiplexing, synchronous circuit

[*]This work is supported by the Renault-Centrale Nantes chair dedicated to the propulsion performance of electric vehicles.

1 Introduction

Timing constraints are a major problem in the design of synchronous logical circuits. To meet these constraints the *pipeline* is often inevitable, it allows increasing the operating frequency and thus the throughput. The circuit composed of atomic operators is sliced in several steps called stages. This slicing, physically implemented with memories (flip-flops), allows the concurrent execution of stages and the synchronization of their inputs/outputs. The automatic generation of a pipeline, *i.e.* the efficient placement of flip-flops (1-bit registers), aims not only at ensuring a target frequency but also minimizing the resources consumed by the pipeline.

1.1 Automatic pipeline generation

The automatic pipeline generation was initially formalized by Leiserson and Saxe in [15], using a model of graph. Their method is based on *retiming*, *i.e.* moving registers in the circuit without altering its behavior. Thanks to *retiming* they are able to build a pipeline guaranteeing a minimal throughput, while minimizing the resources consumed by the pipeline registers [15]. They reformulate the problem with a minimum cost flow problem. But it turned out to be inefficient for large circuits, and therefore has been replaced in [10] by a reformulation into an iterative maximal flow problem. This solution is implemented at the logical synthesis level in the ABC tool [3], which is to our knowledge the current state of the art. However, these approaches are in practice hard to implement, and are mainly used by FPGA vendor tools at the logical synthesis level.

We propose a new approach able to solve this same problem, but also to verify by *model checking* temporal constraints. Moreover, this model, which preserves the structure of the circuit, is quite suitable for optimizations beyond the pipeline, such as the sharing of circuit parts.

1.2 Circuit design with Petri Nets

The authors of [15] introduced an abstraction of circuits as weighted directed graphs. The intuition is actually that the model is a marked graph, which is a subclass of Petri Net (PN) where every place has exactly one input arc and one output arc. Because of their concurrent nature, the PNs have been widely employed to analyze and optimize temporal properties of synchronous and asynchronous circuits: [6, 7, 16, 24].

They proved to be very efficient for latency insensitive systems. Bufistov et al. [6] extended the works of Leiserson and Saxe on latency insensitive systems, by combining *retiming* with *recycling*, *i.e.* insertion of bubbles (registers with no informative value), in order to reduce the total number of registers while ensuring a minimal throughput. More recently, Josipovic et al. [12] proposed a temporal optimization of circuits generated from an HLS (*High Level Synthesis*) description with *control flow* structures, by applying the approach of [6] on extracted sub-circuits.

All those works share the same solving method: deduce temporal constraints from the structure of the PN, and reformulate with a linear optimization problem. In contrast, our approach makes use of the semantics of PNs and synthesizes directly the pipeline from the states of the model. The explicit exploration of the states offers a simple way to verify logical and temporal properties on the resulting pipelined circuit.

1.3 Model-Checking

Model-checking was initially developed, among others, for the verification of electronic circuits [13]. The expressive power of formal models like PN, allows modeling both the circuit and its environment, thus to verify a complete system. We can, for example, verify an FPGA working together with a microcontroler, and all connected to a set of sensors and actuators.

The temporal logics were first introduced by Pnueli [21] as specification languages to describe the behavior of sequential and concurrent systems. The TCTL [2] and weighted CTL [5] logics extend respectively the temporal tree logic CTL, to time and to cost constraints.

We propose to illustrate a usage of those temporal logics for the model-checking of circuits with the example of time-multiplexing. It is a technique of resources sharing using time to sequence the access to a resource. This kind of property can be easily expressed and verified on a PN model.

1.4 Time-Multiplexing

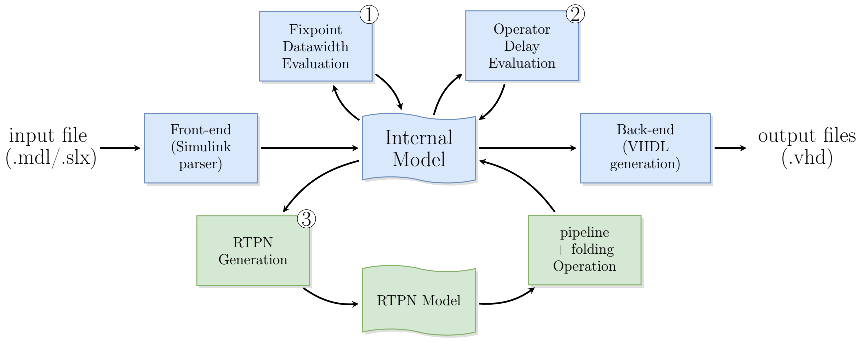
Time-multiplexing is especially interesting for applications with a low throughput with respect to the clock frequency. Such applications include, for example, signal processing applications implemented on FPGA, which require a very low sampling rate compared to the FPGA frequency (*e.g.* a 1 MHz signal processing algorithm onto a 100 MHz FPGA). In this kind of context, it is beneficial to implement only once, parts of the circuit which are used several times, and to schedule their access with the pipeline.

Many works have been carried out in this domain, notably through a problem called *modulo scheduling*. This problem aims for a minimal latency in a time-multiplexed circuit, given limited available resources (arithmetic or logic operators). The authors of [26] proposed an ILP (*Integer Linear Programming*) formulation which combines scheduling constraints, bounds on available resources, minimization of needed registers, and mutualization of registers when it is possible. More recently, a two-step process was demonstrated by [25]: first the shareable configurations are detected, then each configuration is scheduled. This approach called *folding* allows sharing portions of the circuit in contrast with the previous approaches which were only able to share operators one by one.

1.5 Context of application

As part of a collaboration with Renault, the automotive company, we aim at optimizing VHDL synthesis for an FPGA target, from a Matlab/Simulink project. Specifically, the goal is to implement a synchronous circuit with a minimum resource consumption, both for logical units and flip-flops, while ensuring that the whole computation is done in a limited time-frame (the sampling period). A tool is under heavy development, in order to propose a complete chain for the hardware implementation of control laws directly in FPGA. This section briefly introduces the different parts considered in this tool.

The compiler is classically organized around an internal representation, independent of both the input (Simulink) and output (VHDL) representations. This internal model serves as a pivot for all the tools, with different optimizations. The general architecture of the tool is depicted in Figure 11.



Fixpoint Datawidth Evaluation
1
Operator Delay Evaluation
2
input file (.mdl/.slx)
Front-end (Simulink parser)
Internal Model
Back-end (VHDL generation)
output files (.vhd)
RTPN Generation
3
pipeline + folding Operation
RTPN Model

Fig. 1: Simulink-to-VHDL compiler dataflow diagram

The front-end consists in parsing the input file to transform it into this internal model. During this phase, some optimizations are directly performed, such as signal connections between the different modules. In the same way, the back-end (VHDL only at this date) allows generating the hardware description. These output files can be synthesized through the hardware vendor development chain.

Several steps are then performed, each time based on the same internal model and input and output. They allow refining the model. On the diagram presented in Figure 11, only the 3 main steps are shown.

The first pass concerns the evaluation of the size of the signals and the associated encoding (fixpoint). It is currently done manually and is based on a labeling of the signals. This pass can be replaced by the internal tools of Matlab like the toolbox *fixpoint designer*. Eventually, the goal is to make this pass semi-automatic, using both interval arithmetic and affine arithmetic [14], as well as recent work for example for linear looped systems [9].

The evaluation of operator delays is required in modeling with Petri nets extensions defined in the paper. The level of abstraction of the operators in Simulink does not allow to calculate precisely these delays. The approach used, as for the FloPoco tool [8], is to synthesize each operator to determine its critical path. The delay of each evaluated operator is then recorded in a database because this synthesis is computer intensive. This pass is automated through a script and is based on an experimental-only approach. The delay of the operators will necessarily vary during the synthesis of the final project (because of branching, internal optimizations, interconnections, ...). It is only possible to validate that all deadlines are met after the final synthesis.

The last pass (in green in the figure) is the one that is presented in this article and is a synthesis and extension of the articles [18, 20], which introduce the model Timed Petri Net with reset and delayable transitions (RTPN in short) and present its usage for pipeline synthesis. Here, the focus is on the addition of temporal logic constraints to the generation of pipelines.

1.6 Outline

We first present in Section 2 a semantics of Timed Petri Nets "à la Ramchandani" [23] with an atomic firing rule using maximal steps, *i.e.* without the three phases firing. Then, in Section 3, we present an extension of Timed Petri Nets (proposed in [20]) closer to real synchronous circuits, which embeds the impacts of registers on the delay of the circuit with a particular *reset* operation, and which allows relaxing some temporal constraints with *delayable* transitions.

Thanks to this accurate model of pipelined synchronous circuits, we present in Section 4 a design space exploration guided by a cost which stands for the number of needed registers (proposed in [18]). We extend this approach in Section 5 in order to build optimized pipelines (in terms of number of registers), while guaranteeing that a set of properties are verified by the synthesized circuit. We apply this approach to the time-multiplexing problem.

2 A synchronous model for the pipeline

The use of deterministic time in PN was first introduced by Ramchandani [23], which led to a model called Timed Petri Net. Each transition is associated with a delay, representing the fact that actions take time to complete.

$\mathbb{N}$ and $\mathbb{R}_{\geq 0}$ are respectively the sets of integer and non-negative real numbers. For vectors of size n, the usual operators $+, -, <, \leq, >, \geq$ and $=$ are used on vectors of $\mathbb{N}^n$ and $\mathbb{R}^n_{\geq 0}$ and are the point-wise extensions of their counterparts in $\mathbb{N}$ and $\mathbb{R}_{\geq 0}$. As a reminder, for example the point-wise extension of the operator $\leq$ over $\mathbb{N}^n$ is defined by $\forall u, v \in \mathbb{N}^n$, $u \leq v \iff \forall i \in [1..n]$, $u(i) \leq v(i)$. Let $\bar{\mathbf{0}}$ be the null vector of size n.

2.1 The three-phases firing semantics

Ramchandani's semantics is a three-phases firing semantic: delete the input tokens of the transition (consumption), wait until the firing time is reached (delay) and create the output tokens of the transition (production). Once initiated, this firing process cannot be interrupted or stopped. The consumption phase can therefore be seen as a *reservation* (in particular in case of conflict). Moreover, the transitions in the firing process are synchronized to a global clock. He furthermore prohibits zero-time firing, which prevents the same transition from being fired twice when other transitions are in conflict.

Popova proposed a semantics based on the same three-phases firing, but selecting beforehand a maximal step of transitions to fire in the same atomic action [22]. In other words, instead of being reserved one after the other, the transitions are selected and then reserved all at the same time (consumption phase).

2.2 The maximal-step firing semantics

Classically the semantics of (timeless) PN is the interleaving semantics, in which transitions are fired one after the other. In the maximal-step semantics, a maximal set of fireable transitions is selected and are then fired all at once. In practice the maximal-step semantics avoids the interleaving, which is interesting for the modeling of synchronous systems. It imposes more constraints than the interleaving semantics, and thus increases the expressiveness and eliminates reachable markings.

Popova shows how a counter machine can be encoded and simulated by timeless PN with maximal-step firing, which then also applies to TPNs [22]. In particular, she shows the modeling of the so-called zero-test, which is recalled in Figure 1a. It means that Timeless as well as Timed PNs firing in maximal-step are Turing equivalent.

2.3 An atomic semantics for TPNs

We consider a TPNs atomic semantics [20] without any reservation: waiting is done while keeping the tokens in their place, then when at least one transition is fireable we select the maximal step and fire (consumption and production) all the transitions in one atomic action. The maximal step contains enabled transitions which have been enabled for a period of time equal to their delays.

Informally, a clock and a delay are associated with each transition of the Net. The clock measures the time elapsed since the transition has been enabled and the delay is interpreted as a firing condition: the transition may and must fire if the value of its clock is equal to the delay.

Formally:

Definition 1 (TPN). *A TPN is a tuple* $(P, T, {}^{\bullet}(.), (.)^{\bullet}, \delta, M_0)$ *defined by:*

- $P = \{p_1, p_2, \ldots, p_m\}$ *is a non-empty set of* places*,*
- $T = \{t_1, t_2, \ldots, t_n\}$ *is a non-empty set of* transitions*,*

- $^\bullet(.) : T \to \mathbb{N}^P$ *is* the backward incidence function*,*
- $(.)^\bullet : T \to \mathbb{N}^P$ *is* the forward incidence function*,*
- $M_0 \in \mathbb{N}^P$ *is* the initial marking *of the Petri Net,*
- $\delta : T \to \mathbb{N}$ *is the function giving* the firing times *(delays) of transitions.*

A marking M is an element of $\mathbb{N}^P$ such that $\forall p \in P$, $M(p)$ is the number of tokens in place p. A marking M enables a transition $t \in T$ if: $M \geq^\bullet t$. The set of transitions enabled by a marking M is $\mathit{enab}\,(M) = \{t \in T \mid M \geq^\bullet t\}$. A transition is fireable if it is enabled and its clock has reached its delay.

Fireable transitions are fired simultaneously according to the maximal-step firing rule. For a marked graph where every place has one incoming arc, and one outgoing arc, there can not be any conflict and the firing of a transition cannot disable another transition. In the general case, there can be conflict and, from a given state, there can be several maximal steps τ.

From a marking M, the simultaneous firing of a set τ of transitions leads to a marking $M' = M + \Sigma_{t\in\tau}\big(t^\bullet -^\bullet t\big)$.

A transition t' is said to be *newly* enabled by the firing of a set of transitions τ if $M + \Sigma_{t\in\tau}\big(t^\bullet -^\bullet t\big)$ enables t' and $(M - \Sigma_{t\in\tau}\ ^\bullet t)$ did not enable t'. If t remains enabled after its firing then t is newly enabled. The set of transitions newly enabled by a set of transitions τ for a marking M is noted $\uparrow \mathit{enab}\,(M,\tau)$.

A state is a pair (M, v) where M is a marking and $v \in \mathbb{R}^T_{\geq 0}$ is a time valuation of the system (*i.e.* the value of the clocks). $v(t)$ is the time elapsed since the transition $t \in T$ has been newly enabled. $\bar{\mathbf{0}}$ is the valuation assigning 0 to every transition.

Definition 2 (Maximal Step)**.** *Let $q = (M, v)$ be a state of the TPN $(P, T, ^\bullet(.), (.)^\bullet, \delta, M_0)$, $\tau \subseteq T$ is a maximal step from q iff:*

1. $\forall t \in \tau,\ v(t) = \delta(t)$
2. $\sum_{t\in\tau}\ ^\bullet t \leq M$
3. $\forall t' \in T,\ (v(t') = \delta(t')$ *and* $^\bullet t' \leq M$ *and* $t' \notin \tau) \Rightarrow \sum_{t\in\tau}\ ^\bullet t\ +^\bullet t' \nleq M$

The set of maximal steps from q is noted $maxStep(q)$

The first condition ensures that the transitions are ready to fire, *i.e.* the clocks are equal to the delays. The second condition ensures that the transition are fireable, *i.e.* enabled and not in conflict with another transition of τ. The third condition disallows the existence of a proper superset of τ which fulfills the previous two conditions.

The semantics of TPN is defined as a Timed Transition System (TTS). Waiting in a marking is a delay transition of the TTS and firing a maximal step is a discrete transition of the TTS.

Definition 3 (Semantics of a TPN)**.** *The semantics of a TPN is defined by the Timed Transition System $\mathcal{S} = (Q, q_0, \to)$:*

- $Q = \mathbb{N}^P \times \mathbb{R}^T_{\geq 0}$ *is the set of states,*
- $q_0 = (M_0, \bar{\mathbf{0}})$ *is the initial state,*
- $\to \in Q \times (\mathbb{R}_{\geq 0} \cup 2^T) \times Q$ *is the transition relation including a discrete transition and a delay transition.*

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

$$(M,v) \xrightarrow{d} (M,v') \text{ iff } \begin{cases} v'(t) = \begin{cases} v(t)+d & \textit{if } t \in enab(M) \\ v(t) & \textit{otherwise} \end{cases} \\ \forall t \in enab(M),\ v'(t) \leq \delta(t) \end{cases}$$

- *The discrete transition is defined* $\forall \tau \in maxStep\big((M,v)\big)$ *by:*

$$(M,v) \xrightarrow{\tau} (M',v') \text{ iff } \begin{cases} M' = M + \sum_{t\in\tau} \left(t^\bullet - {}^\bullet t\right) \\ v'(t) = \begin{cases} 0 & \textit{if } t \in \uparrow enab(M,\tau) \ \textit{ or } t \notin enab(M') \\ v(t) & \textit{otherwise} \end{cases} \end{cases}$$

A run in a TPN is a sequence $q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \dots$, such that for all i, $q_i \xrightarrow{\alpha_{i+1}} q_{i+1}$ is a transition in the semantics.

In the absence of conflict, the atomic semantics of Definition 3 is equivalent to the three-phases one of Ramchandani (extended with zero firing delay [22]): there exists only one run, there is no indeterminism. In case of conflict, it is possible to construct the three-phases firing in our semantics: just add a zero time transition before each transition, in order to simulate the reservation action [20].

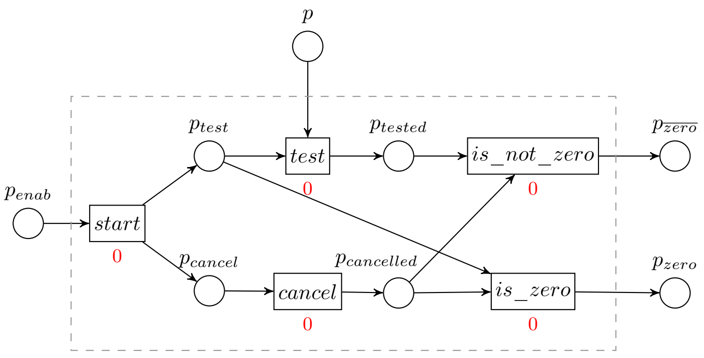
2.4 Zero-test and model of the pipeline

Thanks to the maximal step firing, the TPN represented on Figure 1a can perform the zero-test. The zero-test in the case of TPN is materialized by testing if a place is marked or not, here the place p. After firing the transition *start*, the transitions *test* and *cancel* are fired simultaneously if and only if the place p is marked. The next step will contain the transition *is_zero* iff p wasn't marked, which leads to a token in p_{zero}. We will use this test several times in the following, thus we will replace it by the graphical shortcut of Figure 1b for convenience.

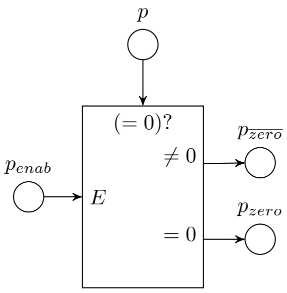
Using this zero-test, it is directly possible to model the dataflow of a pipeline. The TPN of Figure 2 models a D flip-flop, which is used for the synchronization of a signal between two pipeline stages. The marking of the place Q_i (or D_i) represents the presence of data (not its truth value).

On the left of the zero-test, the generator models the oscillator by adding a token in the place *clock* every N time units. When the place *clock* is marked, the zero-test will put a token in Q_i (output) only if the place D_i (input) is marked. We thus model a D flip-flop (copy the input signal D_i to the output Q_i on a rising edge of the clock), which synchronizes the dataflow with *clock*.

Model-checking on pipeline is then possible. However, this model only allows the study of one pipeline (one placement of the D flip-flops in the circuit) at a time. Studying others pipeline requires to change completely the model by moving the flip-flop pattern.



(a) Zero-test pattern



(b) Notation

Fig. 2: Zero-test pattern

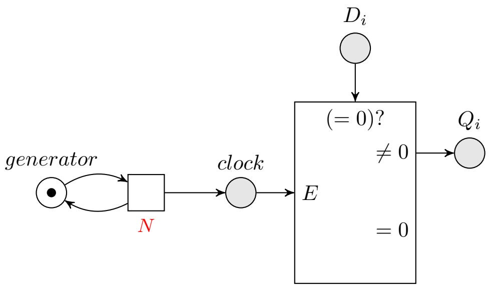


Fig. 3: Model of pipeline dataflow with frequency $\frac{1}{N}$ (D flip-flop)

3 TPN with reset and delayable transitions

We have proposed in [20] an extension of TPN where transitions are separated in two groups: *asap* transitions (non-delayable) must fire as soon as possible, as in Definition 1, and *delayable* transitions can fire when their clock reaches their delay, or when their clock exceeds their delay and they are associated with another transition whose clock just reaches its delay. In addition, the clocks can be resetted (the corresponding action is called *reset*) and the delay between two consecutive *reset*s is fixed by an interval I_{reset}.

3.1 Definitions

Definition 4 (RTPN). *A RTPN $\mathcal{N}$ is a tuple $(P, T, T_D, {}^{\bullet}(.), (.)^{\bullet}, \delta, I_{reset}, M_0)$ defined by:*

- *$(P, T, {}^{\bullet}(.), (.)^{\bullet}, \delta, M_0)$ is a TPN;*
- *$T_D \subseteq T$ is the set of delayable transitions;*
- *I_{reset} is the reset time interval with lower ($\underline{I_{reset}}$) and upper ($\overline{I_{reset}}$) bounds in $\mathbb{N}$.*

From a state (M, v), a transition is fireable if it is enabled and its clock is greater or equal to its delay. As for the TPN, the clock of an *asap* transition $t \notin T_D$ cannot exceed $\delta(t)$. Consequently, $v(t) \leq \delta(t)$ and t must fire when its clock is equal to its delay. A delayable transition $t \in T_D$ may fire either when $v(t) = \delta(t)$ (not delayed in that case), or when $v(t) > \delta(t)$, but in this case t must be associated with at least one (or more) other fireable transition t' such that $v(t') = \delta(t')$.

The maximal step is then maximal only with regard to the *asap* transitions:

Definition 5 (Maximal Step w.r.t. T_D). *Let $q = (M, v)$ be a state of $\mathcal{N}$. A set $\tau \subseteq T$ is a maximal step with regard to T_D from q iff:*

1. *$\forall t \in \tau,\ v(t) \geq \delta(t)$*
2. *$\exists t \in \tau$ s.t. $v(t) = \delta(t)$*
3. *$\sum_{t \in \tau} {}^{\bullet}t \leq M$*
4. *$\forall t' \in T \setminus T_D,\ (v(t') = \delta(t')$ and ${}^{\bullet}t' \leq M$ and $t' \notin \tau) \Rightarrow \sum_{t \in \tau} {}^{\bullet}t + {}^{\bullet}t' \not\leq M$*

The set of maximal steps w.r.t. T_D from q is noted $maxStep_{\setminus T_D}(q)$.

A state is now a pair (M, v) such that $v \in \mathbb{R}_{\geq 0}^{T \cup \{reset\}}$ is extended with a clock value for *reset*, evaluating the time elapsed since the last *reset*. The *reset* action resets all clocks of the net. It is possible only when its clock value belongs to the *reset* interval $v(reset) \in I_{reset}$.

The semantics of a RTPN is defined as a Timed Transition System (TTS). Waiting in a marking is a delay transition of the TTS, and firing a maximal step or the *reset* are discrete transitions of the TTS.

Definition 6 (Semantics of a RTPN). *The semantics of a RTPN $\mathcal{N}$ is defined by the Timed Transition System $\mathcal{S}_{\mathcal{N}} = (Q, q_0, \rightarrow)$ with $Q = \mathbb{N}^P \times \mathbb{R}_{\geq 0}^{T \cup \{reset\}}$ is the set of states; $q_0 = (M_0, \bar{\mathbf{0}})$ is the initial state; and $\rightarrow \in Q \times (\mathbb{R}_{\geq 0} \cup 2^T \cup \{reset\}) \times Q$ is the transition relation including a discrete transition and a delay transition:*

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

$$(M,v) \xrightarrow{d} (M,v') \text{ iff } \begin{cases} v'(t) = \begin{cases} v(t)+d & \text{if } t \in enab(M) \cup \{reset\} \\ v(t) & \text{otherwise} \end{cases} \\ v'(reset) \leq \overline{I_{reset}} \\ \forall t \in enab(M) \setminus T_D,\ v'(t) \leq \delta(t) \end{cases}$$

- *The discrete transition is defined by:*
  - $\forall \tau \in maxStep_{\setminus T_D}\big((M,v)\big)$,

$$(M,v) \xrightarrow{\tau} (M',v') \text{ iff } \begin{cases} M' = M + \Sigma_{t\in\tau}\big(t^\bullet - {}^\bullet t\big) \\ v'(t) = \begin{cases} 0 & \text{if } t \in\uparrow enab(M,\tau) \text{ or } t \notin enab(M') \\ v(t) & \text{otherwise} \end{cases} \end{cases}$$

  - $(M,v) \xrightarrow{\{reset\}} (M,v') \text{ iff } \begin{cases} v(reset) \in I_{reset} \\ v' = \bar{\mathbf{0}} \end{cases}$

Definition 7 (Runs)**.** *Let $\mathcal{N}$ be a RTPN and $\mathcal{S}_{\mathcal{N}}$ its semantics. A run of $\mathcal{N}$ from q is finite or infinite sequence $\rho = q \xrightarrow{d_1} q_{d_1} \xrightarrow{\tau_1} q_{\tau_1} \dots \xrightarrow{d_n} q_{d_n} \xrightarrow{\tau_n} q_{\tau_n}$ of alternating delay d_i (possibly null) and discrete transition τ_i where either $\tau_i \subseteq T$ or $\tau_i = \{reset\}$. For all run ρ, there exists a discrete run $\rho_{\bar{d}} = q \xrightarrow{\tau_1} q_{\tau_1} \dots \xrightarrow{\tau_n} q_{\tau_n}$, in which only the discrete transitions are present.*

3.2 Example

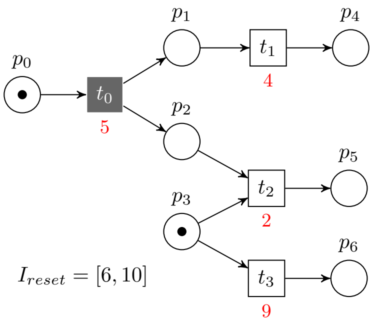
Let us consider the RTPN depicted in Figure 3a. Figure 3b shows a part of its state graph, restricted to the first occurrence of a *reset*. States after a *reset* are framed in cyan. For more convenience, the markings are represented as the set of marked places. The initial state of the net is the state q_0 in the state graph.

Note that the transition t_0 being delayable, it can fire either when it reaches its delay (edge between q_1 and q_2), or together with t_3 (edge between q_8 and q_9). Finally, it should be noted that not all the possible runs are represented here. It is for example possible to wait 7 time units from q_0 and then do a *reset*. Actually it exists infinitely many runs, as a result of the density of time.

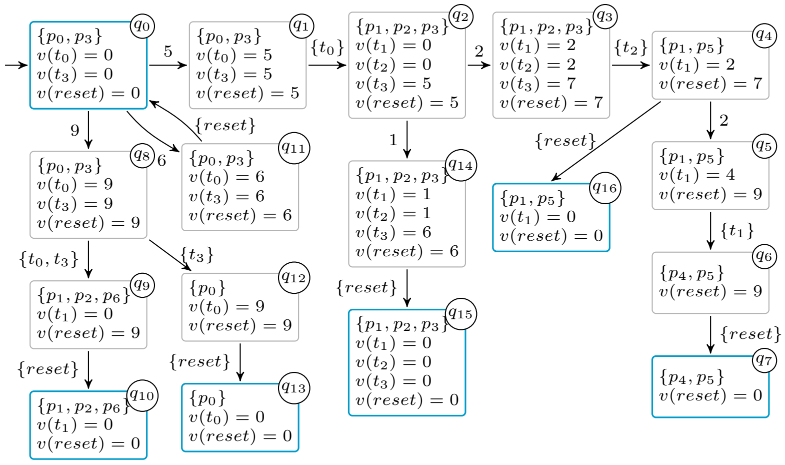
3.3 Properties of RTPNs

Symbolic states

The RTPN semantics is a transition system in which each state consists of a marking and a valuation of the clocks. Note that there is (because we only consider bounded nets) only a finite number of markings, but there are an uncountable quantity of valuations because of the density of time (particularly for the states from which a *reset* is possible). For example, there is infinitely many states between q_8 and q_{11} (in the Figure 3b). These states correspond to the wait between 6 and 9 time units from the state q_0 which can be abstracted



(a) Example of RTPN $\mathcal{N}$



q_0: $\{p_0, p_3\}$, $v(t_0) = 0$, $v(t_3) = 0$, $v(reset) = 0$
5
q_1: $\{p_0, p_3\}$, $v(t_0) = 5$, $v(t_3) = 5$, $v(reset) = 5$
$\{t_0\}$
q_2: $\{p_1, p_2, p_3\}$, $v(t_1) = 0$, $v(t_2) = 0$, $v(t_3) = 5$, $v(reset) = 5$
2
q_3: $\{p_1, p_2, p_3\}$, $v(t_1) = 2$, $v(t_2) = 2$, $v(t_3) = 7$, $v(reset) = 7$
$\{t_2\}$
q_4: $\{p_1, p_5\}$, $v(t_1) = 2$, $v(reset) = 7$
9
6
$\{reset\}$
q_8: $\{p_0, p_3\}$, $v(t_0) = 9$, $v(t_3) = 9$, $v(reset) = 9$
q_{11}: $\{p_0, p_3\}$, $v(t_0) = 6$, $v(t_3) = 6$, $v(reset) = 6$
1
q_{14}: $\{p_1, p_2, p_3\}$, $v(t_1) = 1$, $v(t_2) = 1$, $v(t_3) = 6$, $v(reset) = 6$
$\{reset\}$
q_{16}: $\{p_1, p_5\}$, $v(t_1) = 0$, $v(reset) = 0$
2
q_5: $\{p_1, p_5\}$, $v(t_1) = 4$, $v(reset) = 9$
$\{t_1\}$
$\{t_0, t_3\}$
$\{t_3\}$
q_9: $\{p_1, p_2, p_6\}$, $v(t_1) = 0$, $v(reset) = 9$
q_{12}: $\{p_0\}$, $v(t_0) = 9$, $v(reset) = 9$
$\{reset\}$
q_{15}: $\{p_1, p_2, p_3\}$, $v(t_1) = 0$, $v(t_2) = 0$, $v(t_3) = 0$, $v(reset) = 0$
q_6: $\{p_4, p_5\}$, $v(reset) = 9$
$\{reset\}$
$\{reset\}$
q_{10}: $\{p_1, p_2, p_6\}$, $v(t_1) = 0$, $v(reset) = 0$
$\{reset\}$
q_{13}: $\{p_0\}$, $v(t_0) = 0$, $v(reset) = 0$
q_7: $\{p_4, p_5\}$, $v(reset) = 0$

(b) Part of the state graph of $\mathcal{N}$ (until the first *resets*)

Fig. 4: Example of RTPN and some of its possible runs. Delays are represented in red under the transitions, and delayable transitions are in gray (only t_0 in this example).

by $v(t_0) = v(t_3) = v(reset) \in [6, 9]$. The state space can then easily be abstracted by a finite set of symbolic states (shown in [20]).

Definition 8 (Symbolic state). *A symbolic state is a pair (M, Z) where M is a marking, and the zone Z is a set of valuations v of $T \cup \{reset\}$ defined by the conjunction of:*

- *rectangular constraints: $(v(x) \sim c)$ where $x \in T \cup \{reset\}$, $\sim \in \{\leq, =, \geq\}$ and $c \in \mathbb{N}$,*
- *diagonal constraints: $(v(reset) - v(t) = c)$ where $t \in enab(M)$ and $c \in \mathbb{N}$.*

It is then possible to build a symbolic state graph as defined in [20], illustrated in the example in Figure 4. Compared to the part of state graph of Figure 3b, we can see that some states have been grouped into a single symbolic state.

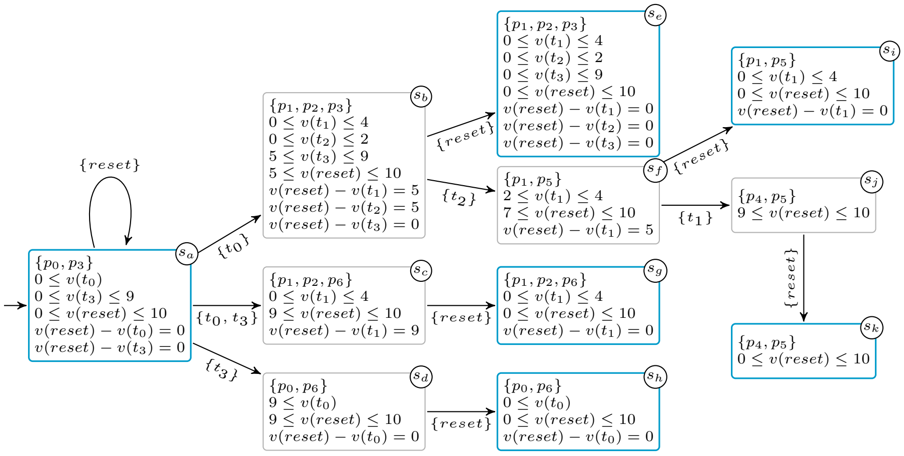

s_a: $\{p_0, p_3\}$, $0 \le v(t_0)$, $0 \le v(t_3) \le 9$, $0 \le v(reset) \le 10$, $v(reset) - v(t_0) = 0$, $v(reset) - v(t_3) = 0$
$\{reset\}$
$\{t_0\}$
$\{t_0, t_3\}$
$\{t_3\}$
s_b: $\{p_1, p_2, p_3\}$, $0 \le v(t_1) \le 4$, $0 \le v(t_2) \le 2$, $5 \le v(t_3) \le 9$, $5 \le v(reset) \le 10$, $v(reset) - v(t_1) = 5$, $v(reset) - v(t_2) = 5$, $v(reset) - v(t_3) = 0$
s_c: $\{p_1, p_2, p_6\}$, $0 \le v(t_1) \le 4$, $9 \le v(reset) \le 10$, $v(reset) - v(t_1) = 9$
s_d: $\{p_0, p_6\}$, $9 \le v(t_0)$, $9 \le v(reset) \le 10$, $v(reset) - v(t_0) = 0$
$\{reset\}$
$\{t_2\}$
s_e: $\{p_1, p_2, p_3\}$, $0 \le v(t_1) \le 4$, $0 \le v(t_2) \le 2$, $0 \le v(t_3) \le 9$, $0 \le v(reset) \le 10$, $v(reset) - v(t_1) = 0$, $v(reset) - v(t_2) = 0$, $v(reset) - v(t_3) = 0$
s_f: $\{p_1, p_5\}$, $2 \le v(t_1) \le 4$, $7 \le v(reset) \le 10$, $v(reset) - v(t_1) = 5$
s_g: $\{p_1, p_2, p_6\}$, $0 \le v(t_1) \le 4$, $0 \le v(reset) \le 10$, $v(reset) - v(t_1) = 0$
s_h: $\{p_0, p_6\}$, $0 \le v(t_0)$, $0 \le v(reset) \le 10$, $v(reset) - v(t_0) = 0$
$\{t_1\}$
s_i: $\{p_1, p_5\}$, $0 \le v(t_1) \le 4$, $0 \le v(reset) \le 10$, $v(reset) - v(t_1) = 0$
s_j: $\{p_4, p_5\}$, $9 \le v(reset) \le 10$
s_k: $\{p_4, p_5\}$, $0 \le v(reset) \le 10$

Fig. 5: Part of the symbolic state graph of $\mathcal{N}$ (until the first *resets*)

It is interesting to note that the zones have a particular shape: the diagonal constraints are equalities and compare all the clocks against $v(reset)$. Thus, by simply setting a value of $v(reset)$ we can “choose” a point of the zone. The discrete actions (other than *reset*) are then only done on integer points of the space and the symbolic state space preserves the language in addition to the reachability. Based on this abstraction it is possible to build a single clock automata that recognized the same language, as presented in [20].

An abstraction preserving branching is readily accessible. Simply split the zone when a transition is no more fireable. For example the state s_a of Figure 4 becomes the states presented in Figure 5.

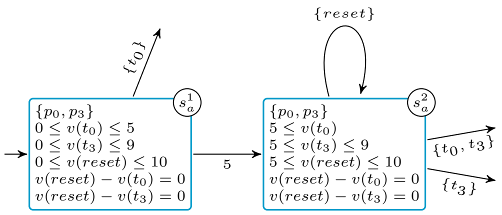

$\{t_0\}$
s_a^1: $\{p_0, p_3\}$, $0 \le v(t_0) \le 5$, $0 \le v(t_3) \le 9$, $0 \le v(reset) \le 10$, $v(reset) - v(t_0) = 0$, $v(reset) - v(t_3) = 0$
5
$\{reset\}$
s_a^2: $\{p_0, p_3\}$, $5 \le v(t_0)$, $5 \le v(t_3) \le 9$, $5 \le v(reset) \le 10$, $v(reset) - v(t_0) = 0$, $v(reset) - v(t_3) = 0$
$\{t_0, t_3\}$
$\{t_3\}$

Fig. 6: Abstraction preserving the branching of state s_a of Figure 4

Without going into details on symbolic state graphs, it should be noted that only few discrete runs are sufficient to describe all the behaviors. We will then only represent the relevant states in the following state graphs.

Decidability and complexity

Timeless PNs, with maximal-step firing, are as expressive as Turing Machines and are a subclass of RTPN for whom the reachability problem is then also not decidable. But if we consider bounded nets, we obtain the following results:

Theorem 1. *Reachability and TCTL model checking for a bounded TPN, with or without reset and delayable transitions, are PSPACE-complete.*

Proof. PSPACE-hardness comes from the PSPACE-completness of the reachability problem for a safe timeless PN with the classical interleaving semantics, which is a subclass of bounded TPN. Then the PSPACE-completness is obtained by applying the same procedure as in [2, 4] by checking TCTL formulae with an inductive algorithm for region graph exploration, which is polynomial in space. □

Theorem 2. *For bounded RTPN, the universality and language inclusion problems are decidable for finite timed words.*

Proof. It is a consequence of the translation preserving the timed bisimulation proposed in [20] of bounded RTPNs to one clock timed automata, for which these problems are decidable [1, 17]. □

4 Pipeline synthesis

The model presented in Section 3 can accurately represent pipelined synchronous circuits. The following will demonstrate how to use this model to build an optimized pipeline which ensures a minimal frequency while minimizing the required registers (and thus the material resource consumed).

In this modeling, transitions represent the operators and places represent the connections of the circuit. The PN is actually a Marked Graph, and there is no conflict. However, the state space still has an exponential size regarding the size of the RTPN. Some features are then used to limit the exploration according to an optimization objective.

The goal is to build the pipeline that minimizes the total number of flip-flops (1-bit registers). The RTPN model is then extended with a cost representing the number of flip-flops of a given pipeline. Note that the considered circuits are finite and with unfolded loops, so we focus on finite runs of RTPN: the accumulation of an increasing cost will not affect the termination.

This pipeline building problem that minimizes the number of registers while ensuring a minimal frequency was already solved by [15]. However, the proposed solution cannot be easily extended with additional constraints on the pipeline. The originality of the approach lies in the possibility to append a set of properties to check, to the pipeline synthesis. Those properties can, for example, permit sharing a portion of the circuit between several pipeline stages. The synthesis of pipeline allowing resource sharing will be addressed afterwards in Section 5.

4.1 RTPN with cost

The RTPN class is extended with a cost on each place, and a marking cost function.

Definition 9 (CRTPN). *A CRTPN is a tuple* $(\mathcal{N}, \mathcal{C}, \omega)$ *where* $\mathcal{N} = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, I_{reset}, M_0)$ *is a RTPN and*

- $\mathcal{C} : P \to \mathbb{N}$ *is the cost assigned to each place;*
- $\omega : \mathbb{N}^P \to \mathbb{N}$ *is the marking cost function (recall that a marking is* $M \in \mathbb{N}^P$*).*

A classical marking cost function is $\omega(M) = \sum_{p \in P} M(p) \cdot \mathcal{C}(p)$ which is the sum of the marking of each place weighted by its cost. This function is not necessarily linear, as we will see for the model of branching points.

Definition 10 (Cost of a run). *The cost* $\Omega(\rho)$ *of a run* ρ *is the accumulated marking cost of the states that immediately follow a reset in the run, starting by the cost of the initial marking.*

It is defined inductively on a run $\rho_n = \rho_{n-1} \xrightarrow{\alpha_n} q_n$*, with* $\alpha_n \in \mathbb{R}_{\geq 0} \cup 2^T \cup \{reset\}$ *and* $q_n = (M_n, v_n)$ *by:*

- $\Omega(q_0) = \omega(M_0)$
- $\Omega(\rho_n) = \begin{cases} \Omega(\rho_{n-1}) + \omega(M_n) & \text{if } \alpha_n = \{reset\} \\ \Omega(\rho_{n-1}) & \text{otherwise} \end{cases}$

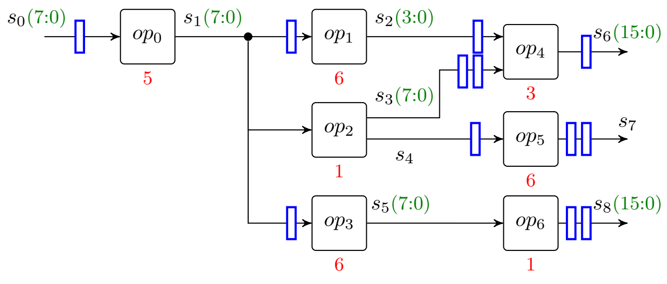
4.2 Modeling circuits

A set of rules to model a circuit with a CRTPN is defined in this section. Figure 6 shows an example of circuit, with the operators op_i connected by signals s_i. The size of each signal is noted between parentheses (in green). The propagation delay of each operator is noted under it (in red). Lastly the pipeline's registers are represented by (blue) rectangles on the edges.

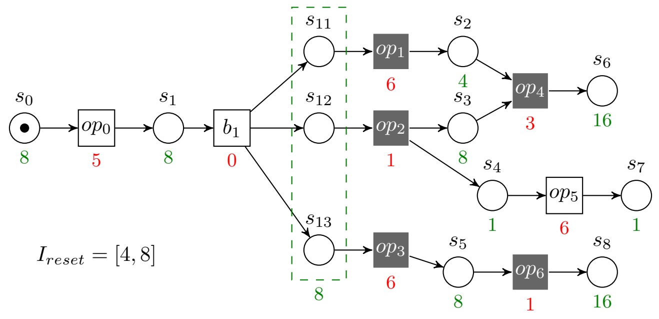
In the following, a circuit is considered as a weighted graph (V, E, d, w), in which $V = Op \uplus B$ is the set of operators Op joint with the set of branching points B ($\uplus$ is the disjoint union), and E is the set of signals. Additional signals following a branching point are defined, in order to match with the edges of the graph (on the example Figure 6 the signal s_1 gives three signals s_{11}, s_{12} and s_{13} after the branching). Finally, the weights d and w represent respectively the propagation delay of the operators $d(op)$ and the size of the signals $w(s)$ (number of bits).

The CRTPN $((P, T, {}^\bullet(.), (.)^\bullet, \delta, I_{reset}, M_0), \mathcal{C}, \omega)$ built from the circuit Figure 6a is represented on Figure 6b, and is obtained thanks to 7 rules. The four first rules preserve the elements of the circuit and their connections:

rule 1: $\exists \phi_e \colon E \mapsto P$ a bijection, with $\forall s \in E$, $\mathcal{C}(\phi_e(s)) = w(s)$;
rule 2: $\exists \phi_v \colon V \mapsto T$ a bijection, with $\forall op \in Op$, $\delta(\phi_v(op)) = d(op)$ and $\forall b \in B$, $\delta(\phi_v(b)) = 0$;
$T = T_{Op} \uplus T_B$ with $T_{Op} = \phi_v(Op)$ and $T_B = \phi_v(B)$.
rule 3: If $s \in E$ is an input signal of $v \in V$, then ${}^\bullet t(p) = 1$ with $t = \phi_v(v)$ and $p = \phi_e(s)$;
rule 4: If $s \in E$ is an output signal of $v \in V$, then $t^\bullet(p) = 1$ with $t = \phi_v(v)$ and $p = \phi_e(s)$;



(a) Pipelined circuit (frequency $f \geq \frac{1}{8}$)



(b) Model of the circuit with a CRTPN

Fig. 7: An example of pipelined circuit and its corresponding model

A place and its associated cost model respectively a signal and its size. A transition and its firing delay model respectively an operator and its propagation delay. Moreover, a transition with a null delay models each branching point (only b_1 in the example). Its purpose is to allow the placement of registers either before the branching (s_1), or on a particular output branch (s_{11}, s_{12} or s_{13}). Rules 3 and 4 preserve the structure of the net.

All the input signals are considered synchronous, which is equivalent to have them all on the first pipeline stage. In the model, this corresponds to the initial marking M_0 defined by rule 5:

rule 5: If s is an input signal (not outgoing from any operator), then $M_0(p) = 1$ with $p = \phi_e(s)$;

The *reset* action models the placement of a border of pipeline stage, and resets all the clocks of the CRTPN for the following stage. Rule 6 defines the upper bound of the reset interval:

rule 6: $\overline{I_{reset}} = \frac{1}{f}$;

The time elapsed since the last *reset* is “stored” in $v(reset)$. The semantics enforces a reset to happen only if $v(reset) \in I_{reset}$. Then, if the upper bound is fixed to $\frac{1}{f}$, the pipeline produced has at least a frequency f. Here $\frac{1}{f}$, and in the following $\frac{1}{2f}$, are supposed to be in $\mathbb{N}$, but they can be rational without altering the results (just apply a factor to all the delays of the model to come back to integer values).

The cost function gives the total number of flip-flops needed in the current pipeline stage:

rule 7: We define $P_{Op} = \{p \in P \mid \exists t \in T_{Op}, t^{\bullet}(p) = 1\}$ and $P_B(p) = \{p' \in P \mid \exists t \in T_B, {}^{\bullet}t(p) = 1 \text{ and } t^{\bullet}(p') = 1\}$.
Then $\forall M \in \{0,1\}^P$, $\omega(M) = \sum_{p \in P_{Op}} \mathcal{C}(p) \cdot \max(M(p), \max_{p' \in P_B(p)}(M(p')))$.

Indeed, the cost of a place matches the size of the signal, and consequently the number of flip-flops needed per register. The cost function manages specifically the branching points, where a mutualization of the registers on the output is possible. That explains why the cost of places after a transition modeling a branching point following a place p, is $\mathcal{C}(p) \cdot \max_{p' \in P_B(p)}(M(p'))$.

These modeling rules are sufficient to fully define the circuit model with CRTPN. All possible pipelines can then be explored thanks to this model. In particular it is possible to find the optimal one, *i.e.* the one that minimizes the resource consumed while ensuring a minimal frequency. However, in practice we quickly face a combinatorial explosion during state space exploration. To avoid combinatorial explosion, the two heuristics proposed in [18] lead to good pratical results by limiting the number of delayable transitions[1] and increasing the lower bound of the *reset* interval.

4.3 Pipeline synthesis from the model

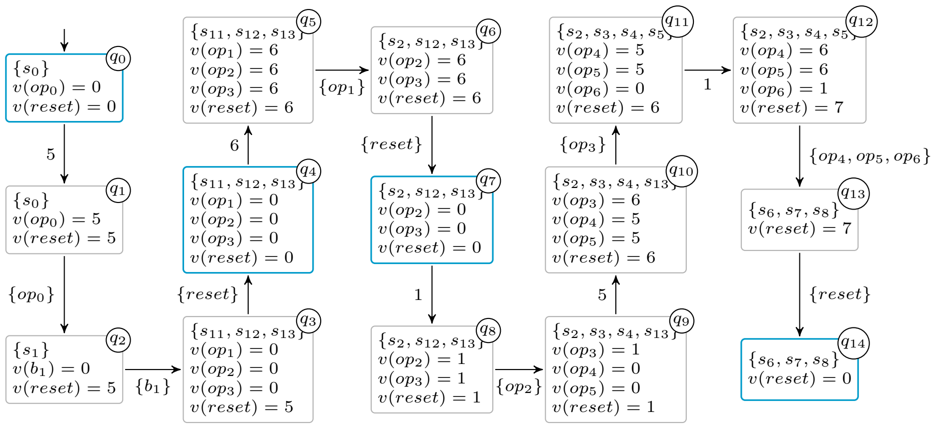
Each reachable state of the model represents a pipeline stage that is possible on the real circuit. The *reset* operation sets the transition from a stage to the following one. The full pipeline is recovered by walking through a branch of the state graph, and accumulating the states following a *reset*.

A run ρ of the CRTPN of Figure 6b, is represented on Figure 7a. It is actually the best achievable run, *i.e.* the one minimizing the cost. The corresponding pipeline in the circuit is drawn on Figure 7b.

Let $q_i = (M_i, v_i)$ $(0 \leq i \leq 14)$ be the states of this run ρ. The marking of each state after a *reset* gives the placement of the registers in the pipelined circuit, except for the signals after a branching point: if several are marked, then only one register is needed in the pipelined circuit (mutualization). For example, the marking $M_4 = \{s_{11}, s_{12}, s_{13}\}$ leads to only one register on s_1.

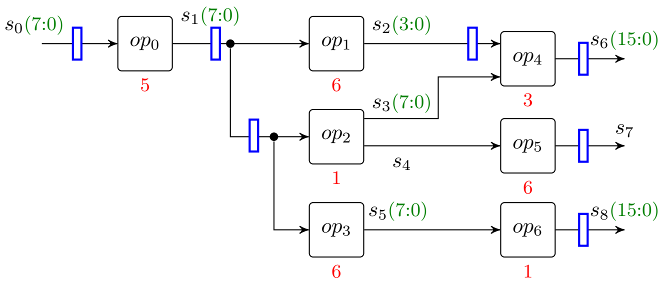
The cost of this run is $\Omega(\rho) = \omega(M_0) + \omega(M_4) + \omega(M_7) + \omega(M_{14}) = \mathcal{C}(s_0)+\mathcal{C}(s_1)+(\mathcal{C}(s_1)+\mathcal{C}(s_2))+(\mathcal{C}(s_6)+\mathcal{C}(s_7)+\mathcal{C}s_8) = 61$. This cost matches the total number of flip-flops in the pipeline of Figure 7b. On this example, a classical greedy "as-soon-as-possible" algorithm as implemented in FloPoCo [11] (a generator of pipelined arithmetic operators for FPGA), produces the example of Figure 6a, with a total of 94 flip-flops (54% more). The improvement of this approach over the classical greedy has been studied in [18], on several arithmetic circuits.

[1]This explains why in the Figure 6b, the transitions modeling the operators op_0 and op_5 are not delayable.



q_0: $\{s_0\}$, $v(op_0) = 0$, $v(reset) = 0$
q_1: $\{s_0\}$, $v(op_0) = 5$, $v(reset) = 5$
q_2: $\{s_1\}$, $v(b_1) = 0$, $v(reset) = 5$
q_3: $\{s_{11}, s_{12}, s_{13}\}$, $v(op_1) = 0$, $v(op_2) = 0$, $v(op_3) = 0$, $v(reset) = 5$
q_4: $\{s_{11}, s_{12}, s_{13}\}$, $v(op_1) = 0$, $v(op_2) = 0$, $v(op_3) = 0$, $v(reset) = 0$
q_5: $\{s_{11}, s_{12}, s_{13}\}$, $v(op_1) = 6$, $v(op_2) = 6$, $v(op_3) = 6$, $v(reset) = 6$
q_6: $\{s_2, s_{12}, s_{13}\}$, $v(op_2) = 6$, $v(op_3) = 6$, $v(reset) = 6$
q_7: $\{s_2, s_{12}, s_{13}\}$, $v(op_2) = 0$, $v(op_3) = 0$, $v(reset) = 0$
q_8: $\{s_2, s_{12}, s_{13}\}$, $v(op_2) = 1$, $v(op_3) = 1$, $v(reset) = 1$
q_9: $\{s_2, s_3, s_4, s_{13}\}$, $v(op_3) = 1$, $v(op_4) = 0$, $v(op_5) = 0$, $v(reset) = 1$
q_{10}: $\{s_2, s_3, s_4, s_{13}\}$, $v(op_3) = 6$, $v(op_4) = 5$, $v(op_5) = 5$, $v(reset) = 6$
q_{11}: $\{s_2, s_3, s_4, s_5\}$, $v(op_4) = 5$, $v(op_5) = 5$, $v(op_6) = 0$, $v(reset) = 6$
q_{12}: $\{s_2, s_3, s_4, s_5\}$, $v(op_4) = 6$, $v(op_5) = 6$, $v(op_6) = 1$, $v(reset) = 7$
q_{13}: $\{s_6, s_7, s_8\}$, $v(reset) = 7$
q_{14}: $\{s_6, s_7, s_8\}$, $v(reset) = 0$
5
$\{op_0\}$
$\{b_1\}$
$\{reset\}$
6
$\{op_1\}$
$\{reset\}$
1
$\{op_2\}$
5
$\{op_3\}$
1
$\{op_4, op_5, op_6\}$
$\{reset\}$

(a) A run of the CRTPN of Figure 6b. The states following a *reset* are framed in cyan (q_0, q_4, q_7 and q_{14})



(b) A possible pipeline of the circuit of Figure 6a

Fig. 8: Example of pipeline synthesized from a run

5 Application to time-multiplexing

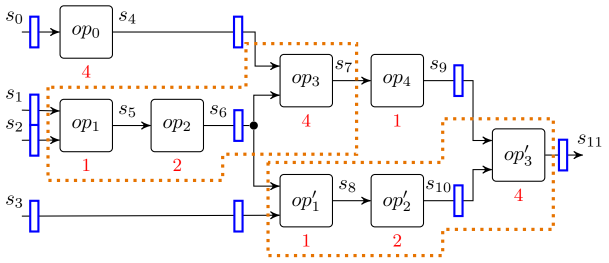
The RTPN model allows exploring many pipelines of a circuit with a frequency guaranteed in an interval. We can check temporal logic properties, such as ensuring that the time spent between the production and consumption of a data on a portion of the circuit is less than a bound. It is in fact possible to impose some specific constraints to the resulting pipelines.

In this section, we apply this approach to the time-multiplexing problem. As stated in the introduction, we focus here on circuits with a low throughput compared to the clock frequency, thus the whole computation is done in one sampling period.

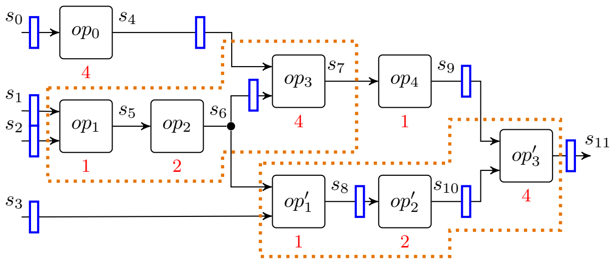
5.1 Folding or time-multiplexing

With an ongoing concern of saving resources, a method called *time-multiplexing* (also called *folding*) has been developed. It aims at sharing the instantiation of operators or group of operators which are needed at several places of the circuit. The sharing is secured by sequential access to the instantiation.

Figure 8 shows an example of circuit that is suitable for resource sharing (based on an example of [27]). Suppose that the operators op_1 (resp. op_2; op_3) and op'_1 (resp. op'_2; op'_3) are two instances of the same operator. It is then possible to instantiate only once the portions of circuit in (orange) dotted frame, and to share the instantiation. Note that the signal sizes are willingly omitted for ease of understanding, but this approach is still valid with different signal sizes (as long as they are equal on the portions to be shared).



(a) Circuit to fold ($f \geq \frac{1}{5}$)



(b) Not foldable pipeline ($f \geq \frac{1}{5}$)

Fig. 9: Example of time-multiplexing

The sequencing of resources access is done with a particular pipeline. Identify this (or one of these) pipeline constitutes the modulo scheduling problem. A key step of modulo scheduling is to find the *initiation interval*: the delay between two introductions of new entries in the circuit. This periodical data introduction can be represented in a RTPN by a token generator on the input. However, as a first step, we will assume that a new data is introduced once the whole computation is done. This assumption is perfectly relevant in signal processing, where the initiation interval (the sampling period) is often significantly higher than the pipeline period (related to the FPGA frequency).

Our goal is to find a particular pipeline on the initial circuit containing all the instances, and afterwards to *fold* the instances (*i.e.* to merge them). The pipeline must then verify two properties:

1. The first is mutual exclusion. The resources must not be accessible at the same time by several clients.
2. The second concerns the register placement. The registers must be placed on the same locations in the portions of circuit to fold.

In the example in Figure 8, the first constraint ensures that there is never some data at the same time on the *twin* signals: simultaneously in s_5 (resp. s_6) and in s_8 (resp. s_{10}). The second constraint means that there must be the same amount of registers in the pair of twin signals. That is why the pipeline of Figure 8b doesn't allow time-multiplexing, contrary to the one of Figure 8a. Indeed, s_5 crosses no register whereas s_8 crosses one, which violates the second constraint.

It is possible to build these particular pipeline, using the approach presented in Section 4, by guiding the exploration with CTL properties. Thus, the exploration will be restricted to runs verifying those properties. Mutual exclusion can be simply expressed as a marking constraint verified by all states of the run. However, the CTL properties are expressed on markings, they do not allow observing the *reset* fired, which is required by the second constraint. As the *reset* is defined in the semantics of the model, it is not explicitly present in the net like the other transitions, it then cannot be connected to an observer place. Somehow the CTL must be extended with the ability to capture the *reset*s fired in each place.

5.2 Explicit reset

As explained in Section 3, due to the density of time the *reset* can fire from an infinite number of states with the same marking, but the successor state will always be the same. Therefore, the only relevant firing times either correspond to the bounds of the interval ($\underline{I_{reset}}$ and $\overline{I_{reset}}$) or are grouped together with a maximal step (in the semantics just after the maximal step firing). One can then consider the *reset* as a delayable transition with a temporal upper bound, and this preserves the discrete runs. Thus, for all RTPN, a TPN with delayable transitions that verifies the same CTL properties can be built.

Indeed, a *reset* can be explicitly expressed with the pattern drawn in Figure 9. The place p_{reset} holds a token since the last firing of *reset*. The delayable transition with delay $\underline{I_{reset}}$ and the non-delayable transition with delay $\overline{I_{reset}}$ model the *reset* interval. For all places p_i of the net, the pattern in the dashed frame is added and connected to the transitions *reset* and *end_reset*. This pattern achieves the *reset* of the transitions enabled by the place p_i. It works in two steps: first the tokens in the place are drained and temporarily held in p_i^{stock}, then once the draining is finished all tokens are replaced in the place p_i. Each step is based on a zero-test with a loopback which simulates a *while loop*. In other words, the tokens are removed from p_i (resp. p_i^{stock}) while there are some left. Note that with a safe net, the pattern can be greatly simplified: only one zero-test that removes and sets back the token in p_i is enough.

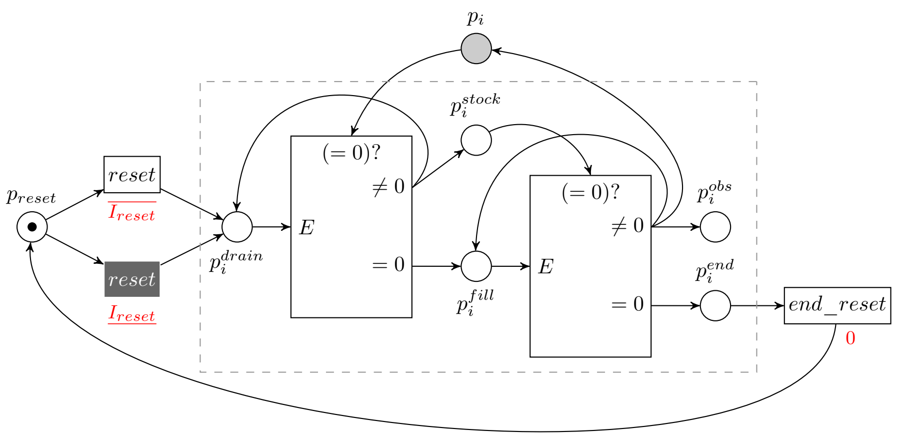


Fig. 10: Pattern expressing the *reset*

In the pattern drawn in Figure 9, the place p_i^{obs} counts the total number of tokens which underwent a *reset* in p_i since the initial state. This observer place allows us to extend the temporal logic CTL for RTPN.

Definition 11. *Let $\mathcal{N}$ be an RTPN, and $q = (M, v)$ a state of $\mathcal{N}$. Let the property $\phi = (reset(p_i) \sim n)$ with $\sim \in \{<, >, \leq, \geq, =, \neq\}$ and $n \in \mathbb{N}$. The state q verifies the property ϕ if and only if q verifies the property $\psi = ((M(p_i^{obs}) \sim n)$.*

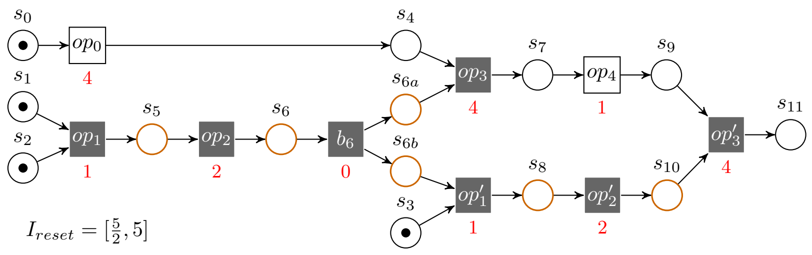
5.3 Synthesis of a pipeline for the folding

With the RTPN approach and the CTL extension presented previously, it is possible to solve the modulo scheduling problem for the folding of circuits. More specifically, it is possible to build a pipeline allowing folding, while ensuring a minimal frequency and minimizing the number of flip-flops. This solution is applied to the example of circuit given in Section 5.1.

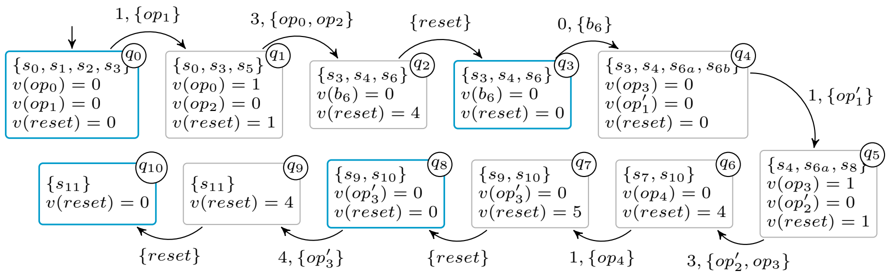
Our method for solving the folding problem is a reachability problem, where a CTL property define mutual exclusion. This property allows the state space exploration to be pruned on the fly. Thus, the synthesis is guided by the CTL property.

The RTPN model of the circuit of Figure 8 is built using the modeling rules presented in Section 4. It is represented in Figure 10a. Note that the transitions modeling the shared operators are delayable in order to relax the exploration, and so to satisfy the folding constraints. The places drawn in orange (s_5, s_6, s_{6a}, s_{6b}, s_8 and s_{10}) are subject to CTL constraints.

A first atomic property guarantees the mutual exclusion of data in the twin places: $\phi_{mutex} = (M(s_5) + M(s_8) \leq 1) \wedge (M(s_6) + M(s_{10}) \leq 1) \wedge (M(s_{6a}) + M(s_{10}) \leq 1) \wedge (M(s_{6b}) + M(s_{10}) \leq 1)$. In fact, in the model, tokens represent both the placement of the future registers, and the data propagated when the circuit is pipelined (as soon as pipeline registers are in place). The property ϕ_{mutex} checks that



(a) RTPN


$1, \{op_1\}$
$3, \{op_0, op_2\}$
$\{reset\}$
$0, \{b_6\}$
$1, \{op'_1\}$
q_0: $\{s_0, s_1, s_2, s_3\}$ $v(op_0) = 0$ $v(op_1) = 0$ $v(reset) = 0$
q_1: $\{s_0, s_3, s_5\}$ $v(op_0) = 1$ $v(op_2) = 0$ $v(reset) = 1$
q_2: $\{s_3, s_4, s_6\}$ $v(b_6) = 0$ $v(reset) = 4$
q_3: $\{s_3, s_4, s_6\}$ $v(b_6) = 0$ $v(reset) = 0$
q_4: $\{s_3, s_4, s_{6a}, s_{6b}\}$ $v(op_3) = 0$ $v(op'_1) = 0$ $v(reset) = 0$
q_5: $\{s_4, s_{6a}, s_8\}$ $v(op_3) = 1$ $v(op'_2) = 0$ $v(reset) = 1$
q_6: $\{s_7, s_{10}\}$ $v(op_4) = 0$ $v(reset) = 4$
q_7: $\{s_9, s_{10}\}$ $v(op'_3) = 0$ $v(reset) = 5$
q_8: $\{s_9, s_{10}\}$ $v(op'_3) = 0$ $v(reset) = 0$
q_9: $\{s_{11}\}$ $v(reset) = 4$
q_{10}: $\{s_{11}\}$ $v(reset) = 0$
$\{reset\}$
$4, \{op'_3\}$
$\{reset\}$
$1, \{op_4\}$
$3, \{op'_2, op_3\}$

(b) A run allowing the folding

Fig. 11: Time-multiplexing with RTPN

the two signals s_5 (resp. s_6, s_{6a}, s_{6b}) and s_8 (resp. s_{10}) will never contain data simultaneously.

A second atomic property guarantees the consistency of registers placement in the twin places in a final state (state with the final marking): $\phi_{consist} = (M(s_{11}) = 1) \wedge (reset(s_5) = reset(s_8)) \wedge (reset(s_6) + reset(s_{6a}) = reset(s_{10})) \wedge (reset(s_6) + reset(s_{6b}) = reset(s_{10}))$. It checks that in a final state $(M(s_{11}) = 1)$, the twin places have passed through the same number of *resets*.

The final CTL property guarantees that ϕ_{mutex} holds until it is satisfied together with $\phi_{consist}$ (once a final state is reached): $\phi_{fold} = \mathbf{A}(\phi_{mutex}\mathbf{U}(\phi_{mutex} \wedge \phi_{consist}))$. The pruning during the state space exploration insures that each run satisfies the property ϕ_{fold}, as the one presented in Figure 10b. The synthesized pipeline from this run is the one drawn in Figure 8a.

5.4 Actually fold the circuit

The parts of shareable circuits are currently selected by hand. This problem has been addressed in the literature as automatic identification of isomorphic subgraphs, and some solutions have already been proposed [27]. In our case study, we benefit from the high-level abstraction of Matlab/Simulink and some parts of the evaluated models have fairly obvious redundancies to be determined. On the other hand, the use of libraries facilitates the identification of shareable parts.

Once the portions to be shared have been chosen, a minimum bound on the pipeline frequency f_p can be deduced from the desired minimum sampling rate f_s. Let n be the maximum number of times a circuit portion is shared, then the pipeline frequency must satisfy $f_p \geq n \cdot f_s$. This bound generalizes to the case of sharing nested parts, by multiplying the maximum number of occurrences between nested levels. However, this bound does not guarantee that the resulting folded circuit will have the minimum sampling rate f_s. A post-selection of the produced pipelines, based on their latency (which is the sampling period), is therefore necessary. In other words, the lower bound on the pipeline frequency f_p is only used to reduce the exploration to a few potential solutions.

Our pipeline synthesis approach for folding has been implemented in a prototype tool. The first tests are encouraging, and on the example presented, the tool produces the expected pipeline in a few milliseconds: 24ms on an Intel Core i7-6700HQ processor.

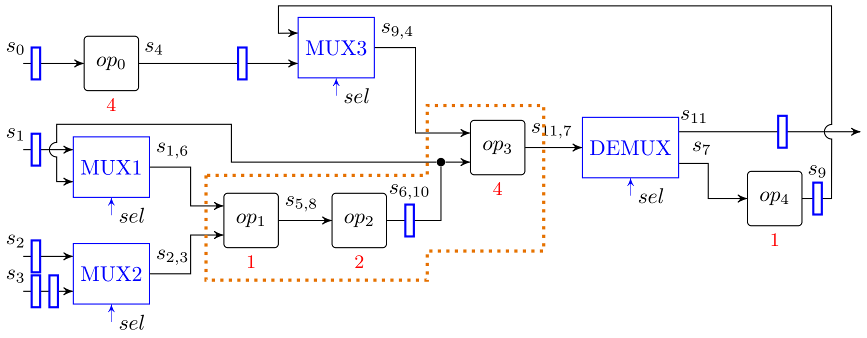


Fig. 12: Circuit defined in Figure 8a after folding

The last step is to generate the operators effectively folding the circuit. To do so, the shareable parts are merged, and multiplexers (MUX) are added in the circuit in place of their input/output registers. Figure 12 shows the folded circuit obtained from the pipelined circuit of Figure 8a. Multiplexers (and demultiplexers) are associated with a control signal produced by a sequencer (*sel* on the figure), that selects the input (output) signals. This part is under development since the code generation associated with the folding step is not yet implemented.

6 Conclusion

We proposed a model for the pipeline of synchronous circuit based on TPN “à la Ramchandani”. Although it allows verification, this model is only able to study one pipeline at a time.

We then focused on the optimization of the pipeline in terms of resources. The RTPN model allows generating multiple pipelines with a target frequency, and to select one minimizing the resource consumption with regard to the size of registers and their eventual mutualization.

Using this result we then concentrated on handling together the optimization with the synthesis of pipeline following a specification expressed in temporal logic. In particular, we presented a solution to the time-multiplexing problem. It relies on the explicit expression of the *reset* in the RTPN model. This method produces a pipeline allowing time-multiplexing, ensuring a target frequency, and minimizing the number of flip-flops.

While these results are encouraging, their implementation may be slowed down by combinatorial explosion. We consider the possibility to combine them with an ILP approach in order to obtain both the computation speed and the possibilities offered by model-checking. We also think about exploiting further model-checking towards the interactions with the environment.

Declarations

The authors declare that they have no conflict of interest.

References

[1] Parosh Aziz Abdulla, Johann Deneux, Joël Ouaknine, Karin Quaas, and James Worrell. Universality analysis for one-clock timed automata. *Fundam. Informaticae*, 89(4):419–450, 2008.

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

[3] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, release 70930.

[4] Hanifa Boucheneb, Guillaume Gardey, and Olivier H. Roux. TCTL model checking of time Petri nets. *Journal of Logic and Computation*, 19(6):1509–1540, December 2009.

[5] Patricia Bouyer, Kim Guldstrand Larsen, and Nicolas Markey. Model checking one-clock priced timed automata. *Logical Methods in Computer Science*, 4(2), May 2008.

[6] D. Bufistov, J. Cortadella, M. Kishinevsky, and S. Sapatnekar. A general model for performance optimization of sequential systems. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007.

[7] J. Campos, G. Chiola, J. M. Colom, and M. Silva. Properties and performance bounds for timed marked graphs. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 39(5):386–401, 1992.

[8] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

[9] Thibault Hilaire and Anastasia Volkova. Error analysis methods for the fixed-point implementation of linear systems. In *2017 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 1–6, 2017.

[10] A. P. Hurst, A. Mishchenko, and R. K. Brayton. Fast minimum-register retiming via binary maximum-flow. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 181–187, 2007.

[11] Matei Istoan and Florent de Dinechin. Automating the pipeline of arithmetic datapaths. In *Design, Automation & Test in Europe Conference & Exhibition (DATE 2017)*, pages 704–709, Lausanne, Switzerland, 2017.

[12] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. Buffer placement and sizing for high-performance dataflow circuits. In *Proc. of the 2020 ACM/SIGDA Int. Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 186–196, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4, April 1999.

[14] D. U. Lee, A. A. Gaffar, R. C.C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-guaranteed bit-width optimization. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 25(10):1990–2000, October 2006.

[15] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, June 1991.

[16] Mehrdad Najibi and Peter A. Beerel. Slack matching mode-based asynchronous circuits for average-case performance. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, page 219–225. IEEE Press, 2013.

[17] J. Ouaknine and J. Worrell. On the language inclusion problem for timed automata: closing a decidability gap. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 54–63, 2004.

[18] Rémi Parrot, Mikaël Briday, and Olivier H. Roux. Pipeline Optimization using a Cost Extension of Timed Petri Nets. In *The 28th IEEE International Symposium on Computer Arithmetic (ARITH 2021)*. IEEE, June 2021.

[19] Rémi Parrot, Mikaël Briday, and Olivier H Roux. Réseaux de Petri temporisés pour la conception et vérification de circuits pipelinés. In *Modélisation des Systèmes Réactifs (MSR'21)*, Paris, France, November 2021.

[20] Rémi Parrot, Mikaël Briday, and Olivier H. Roux. Timed Petri Nets with Reset for Pipelined Synchronous Circuit Design. In *The 42th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2021)*, volume 12734 of *Lecture Notes in Computer Science*. Springer, June 2021.

[21] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA*, pages 46–57. IEEE Computer Society, 1977.

[22] Louchka Popova-Zeugmann. *Time and Petri Nets*. Springer, 2013.

[23] C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge,

MA, 1974.

[24] Sangyun Kim and P. A. Beerel. Pipeline optimization for asynchronous circuits: complexity analysis and an efficient optimal algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):389–402, 2006.

[25] P. Sittel, N. Fiege, M. Kumm, and P. Zipf. Isomorphic subgraph-based problem reduction for resource minimal modulo scheduling. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2019.

[26] P. Sittel, M. Kumm, J. Oppermann, K. Möller, P. Zipf, and A. Koch. Ilp-based modulo scheduling and binding for register minimization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 265–2656, 2018.

[27] Patrick Sittel, Konrad Möller, Martin Kumm, P. Zipf, Bogdan Pasca, and Mark Jervis. Model-based hardware design based on compatible sets of isomorphic subgraphs. 12 2017.