Toward the Formal Verification of HILECOP: Formalization and Implementation of Synchronously Executed Petri Nets

Vincent Iampietro, David Andreu, David Delahaye

Toward the Formal Verification of HILECOP: Formalization and Implementation of Synchronously Executed Petri Nets

Vincent Iampietro[1], David Andreu[1,2], and David Delahaye[1]

[1] LIRMM, Université de Montpellier, CNRS, Montpellier, France
[2] NEURINNOV, Montpellier, France
Firstname.Lastname@lirmm.fr

Abstract. The HILECOP methodology is a process for the design of critical digital systems. In HILECOP, Petri Net (PN) models are used as a high-level formalism to specify the behavior of the designed systems. VHDL (VHSIC Hardware Description Language) code is then automatically generated from PN models to implement the digital system on Field Programmable Gate Array (FPGA) circuits. The goal of this work is to formally verify that through this model-to-text transformation, the behavior described by a PN model is preserved in the produced VHDL code, knowing that the transformed PN models are synchronously executed on the target. As a first step toward the achievement of this goal, we present our implementation of HILECOP's PN structure and semantics, which has been formalized using the Coq proof assistant. We also describe a token player program for these PNs, which has been proved sound and complete with respect to HILECOP's PN semantics.

Keywords: Critical Digital Systems, Formal Verification, Petri Nets, HILECOP Methodology, Coq Proof Assistant

1 Introduction

The safe design of critical digital systems has been widely investigated but remains relevant. The use of formal models is mandatory, with the help of their mathematical foundations, to assess the soundness of the design. However, this formalization rarely provides guarantees beyond the design itself. The formal model is frequently hand-coded by the engineer, sometimes with automatic generation of a code skeleton to be completed. While automatic generation facilitates the task and limits the risk of error, it remains to be proven that the properties highlighted on the design model are preserved on its implemented version, taking into account the impact of its execution on the target.

Although some work has been done in the field of software to automatically generate a code with guaranteed properties from an abstract model (for instance, see the B method [1]), this is less common in the field of electronic programmable components. This is exactly the context of our work, and more precisely, we aim

to produce active implantable medical devices from FPGA-type components using a reliable methodology.

Our approach relies on the HILECOP methodology [8]. This methodology allows us to design digital systems using formal models with a high level of abstraction. The formalism of these models makes it possible to specify both architecture and behavior in a formalism that provides structuring, exception-handling, reuse, analysis, as well as readability to facilitate discussions between engineers around the design. These formal models are then transformed to VHDL code [15], and finally synthetized on FPGA circuits.

To get a reliable methodology, we have to verify that the transformations carried out by HILECOP in the process of producing concrete code preserves the semantics of the initial models. To do so, we propose to formally prove this semantic preservation property and mechanize the corresponding proof with the help of the Coq proof assistant [14].

To achieve the formal verification of HILECOP in Coq, our approach is similar to what has been done in the context of verification of programming environments (see [10], for instance). The idea is to formalize the semantics of the source and target languages, and verify that the transformation preserves the semantics of any input model.

In the case of HILECOP, some specificities of the source and target languages introduce additional technical difficulties in the process of formal verification. A first difference concerns HILECOP's high-level formalism (the input language), which is quite abstract. This formalism depends on PNs, and thus is not a common programming language. Moreover, due to constraints coming from the final implementation on FPGA circuits, PNs have to be synchronously executed. This is also very specific as PNs are usually executed in an asynchronous way.

A second difference is about the VHDL language (the output language). Similarly to the PN models used in HILECOP, the VHDL language is not a common programming language as its purpose is both the structural and behavioral description of hardware circuits. Although previous work has been conducted toward the formalization of the VHDL semantics [7], a semantics that is able to both handle all the constructs in the generated programs, and facilitate the proof of behavior preservation, still needs to be designed.

The process of formally verifying the entire HILECOP transformation chain is a long way to go. In this paper, we present a first step, which consists in modeling the behavioral part of the input language, i.e. the part involving PNs. The modeling consists of the definition of the PN structures, as well as their synchronous evolution semantics. Both are implemented using the Coq proof assistant. In addition, to increase confidence in our implementation of the PNs semantics, we develop a token player program, which is proved to be sound and complete with respect to the PN semantics.

To the best of our knowledge, there is very little work on PN implementation and semantics in the context of formal and mechanized proofs. In [5], a Coq implementation of place/transition PNs is thoroughly described. The goal of the implementation is to get an automated proof that a given PN is a refinement

of another. The refinement relation between two PNs is a structural equivalence relation, and no implementation of the place/transition PN semantics is given in this work.

Literature regarding the verification of model-to-model transformation is divided in two approaches: the verification of structural property preservation, and the verification of semantics preservation through model transformation. Concerning the verification of structural property preservation, in [4], the authors use the Coq proof assistant to specify metamodels, models, and transformation rules from source to target (meta)models. The framework allows us to complete proofs that, given some pre-conditions on the structure of source models, the transformation rules entail necessary post-conditions regarding the structure of target models. In [2] and [11], verification of structural property preservation is realized over model transformation involving PNs. However, in these papers, PNs are only given as an illustrating example of models, and the authors are not interested in a thorough implementation of PNs and their semantics, since the proofs only deal with structural preservation. But to show the soundness of the code generation in HILECOP, we will also need to consider this aspect of structural equivalence between the initial PN models and the corresponding VHDL designs, even though this will probably be the easiest part of the proof.

The other aspect of work regarding the verification of model-to-model transformation deal with semantics preservation. In this domain, literature provides a lot of work on the formal verification of compilers with the Coq proof assistant [10,13,3]. Other work focuses on models rather than programming languages. For example, in [6], the authors deal with the verification of transformation from Colored Petri nets (CPNs) to Low Level Virtual Machine (LLVM) programs. More precisely, the authors formalize the CPN semantics through the use of two functions computing the graph of reachable states, and also describe a formal operational semantics for the LLVM language. Finally, the authors provide a proof that the generated LLVM program encodes the corresponding graph of reachable states. In [16], the authors aim to verify a transformation from Architecture Analysis and Design Language (AADL) models, which is a formalism to design systems at hardware and software levels, to Time Abstract State Machines (TASMs), which is a formalism to express concurrent systems. In this paper, a formal semantics for AADL models and TASMs is expressed in terms of Timed Transition Systems (TTSs). The authors then proves a theorem of strong simulation equivalence between the TTSs representing AADL models and the ones representing TASMs, which are generated through a model transformation. All the contributions if this paper have been mechanized using the Coq proof assistant.

The paper is organized as follows: Sec. 2 gives an overview of the HILECOP methodology; we then introduce, in Sec. 3, the PNs that are used in the input language of HILECOP; next, in Sec. 4, we present the formalization and implementation of these PNs, which has been realized in Coq; finally, in Sec. 5, we describe, still in Coq, the development of a token player, together with its soundness and completeness proofs with respect to the PN semantics.

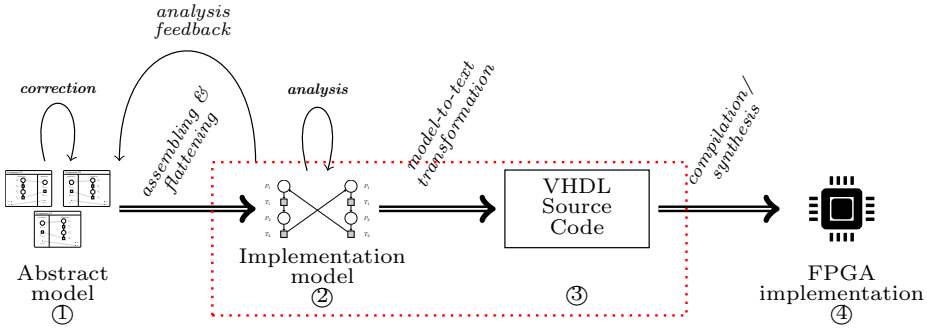


Fig. 1: Workflow of the HILECOP Methodology

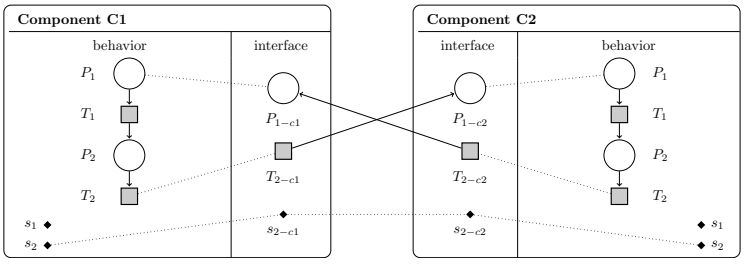


Fig. 2: An Example of HILECOP Model

2 The HILECOP Methodology

The HILECOP methodology consists of a process for the design and implementation of critical digital systems. This methodology relies on several transformations going from abstract models to concrete FPGA implementations through the production of VHDL code. Fig. 1 details the global workflow of HILECOP. In this figure, Step 1 corresponds to the model of a digital system. This model is built with component diagrams and the behavior of each component is described by means of PNs. Fig. 2 provides an example of such model. As shown in this figure, an internal behavior and an interface outline the structure of components. The component interface exposes places, transitions, and signals, which are references to nodes of the internal behavior, from which the components can be assembled to get the global behavior of the digital system.

Next, in Fig. 1, the transformation from Step 1 to Step 2 flattens the model. The internal behaviors are connected according to the interface compositions, and embedding component structures are removed. The analysis phase, going from Step 2 to Step 1, aims to produce a model that is conflict-free (see Sec. 3.2 for more details about the definition of a conflict), bounded, and deadlock-free, using model-checking techniques. After several iterations, the model should reach soundness and is then said to be implementation-ready.

From Step 2 to Step 3, VHDL source code is then generated by means of a model-to-text transformation. This generated code describes the hardware system that will be implemented in a FPGA circuit. From Step 3 to Step 4, the VHDL compilation/synthesis and the FPGA programming are finally performed using industrial tools.

In this work, we focus on the part of the workflow in Fig 1 that is framed with dotted lines, i.e. the model-to-text transformation between Step 2 and Step 3. In particular, we aim to prove that through this model-to-text transformation, the behavior described by the initial model is preserved in the generated VHDL code. To do so, we have to implement the structure of PNs used in the HILECOP models, together with the semantics providing the evolution rules of these PNs.

3 HILECOP's Petri Nets

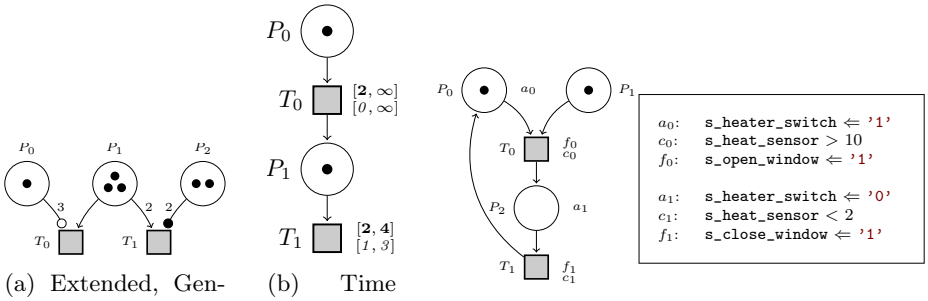
The PNs that are used to describe the behavior of HILECOP components are Synchronously executed, extended, generalized, Interpreted, and Time Petri Nets with priorities and macro-places (SITPNs) [8]. The formalism was created to deal with multiple problems at the same time: the need for model analysis, the need for a deterministic semantics for the models since we have to consider the design of safety-critical systems, and finally, the need for a synchronous semantics motivated by the final implementation of the models as hardware circuits that are synchronized with a clock signal.

In this paper, we present SITPNs but without macro-places. In this section, we assume that the reader is familiar with the basic PN formalism, and we describe the evolution mechanism of SITPNs.

3.1 Extended, Generalized, Interpreted, and Time Petri Nets

SITPNs are generalized PNs in the sense that they admit edge weights to be natural numbers. They are also extended in the sense that they introduce new kinds of edges, i.e. inhibitor and test edges, respectively represented by edges with white and black circle heads. Fig. 3.a shows an example of generalized and extended PN.

SITPNs are also time PNs, where time intervals are associated to transitions. As can be seen in Fig. 3.b, time intervals are of the form $[a, b]$, with $a \in \mathbb{N}^*$ and $b \in \mathbb{N}^* \sqcup \{\infty\}$, where $\mathbb{N}$ is the set of natural numbers (non negative integers). When a transition is sensitized, the two bounds of its time interval are decremented at each time step (a clock cycle in our case, see Subsec. 3.2). When the lower bound is equal to zero, then the transition has reached the firing time window and must be fired right away (the semantics of time intervals is imperative). In Fig. 3.b, the time intervals in bold are the static time intervals associated with transitions at the PN initialization, while the time intervals in italics are dynamic time intervals (when the PN is executed). A time interval is reset when an associated transition is disabled.



(a) Extended, Generalized Petri Net (b) Time Petri Net (c) Interpreted Petri Net

Fig. 3: Interpreted and Time Petri Nets

SITPNs also include the notion of interpreted PNs, which is useful to express the interaction between a system and its environment. This notion of interpretation introduces three new concepts:

- Continuous actions, associated to places: actions associated to a place p are activated as long as p is marked;
- Functions (or discrete actions), associated to transitions: when a transition t is fired, all functions associated to t are executed;
- Conditions, associated to transitions: conditions are Boolean expressions whose values depend on the environment.

As functions and actions can be any kind of operations that have an impact on the environment of the system, their effects are measured through the values of conditions attached to the transitions of the system. The condition value controls the firing of the transition to which it is attached (no firing if the condition is false). Fig. 3.c illustrates the use of actions, functions, and conditions in an interpreted PN. In this figure, the action a_0 is activated as the place P_0 is marked. The function f_0 will also be executed at the firing of T_0, which depends on the value of the condition c_0. The table of Fig. 3.c presents some VHDL code associated to actions, functions, and conditions, but the expression of interpretation goes beyond the scope of VHDL.

In generalized, extended, interpreted, and time PNs, a transition is firable if it is sensitized, its conditions are true, and its time interval is ready.

3.2 Synchronous Execution and Conflict Resolution

The SITPN state evolution is synchronized with two clock events of a clock signal, namely the falling and rising edges of the signal, as represented in Fig. 4. The falling edge event, i.e. the event ① in Fig. 4, triggers the update of the value of conditions, the activation of the actions associated to marked places, the update of the time intervals of transitions, and the selection of the transitions to be fired. The marking is then updated on the rising edge event, i.e. the event ② in Fig. 4, so that all transitions to be fired are fired at once, and the

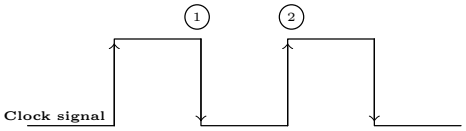


Fig. 4: Evolution of a SITPN Synchronized with a Clock Signal

functions associated to fired transitions are then executed. Moreover, the rising edge determines which transitions have been disabled in a transitory manner and must have their time intervals reset. Also, all the time intervals that have reached their upper bound value are marked as blocked. The case of a blocked time interval is possible, despite of the imperative semantics associated to the firing of time transitions, i.e. transitions with a time interval. If a time transition is bound to a condition that is always false, then it cannot be fired even though its time interval is ready, and as long as this situation holds, the time interval is decremented at each clock cycle, until it eventually gets blocked.

The semantics of synchronous execution requires that all transitions are fired at the same time, while in asynchronous PNs, transitions are fired one at a time. However, we cannot allow two transitions sensitized by the same token to be fired at the same time, since we must preserve the balance between resource consumption and transition firing. Transitions that have a common input place are in structural conflict. Moreover, transitions are in effective conflict if they are in structural conflict and if the firing of one of them disables the others. To solve effective conflicts, priorities are introduced between transitions and allow us to compute a firing order in case the marking would not provide enough tokens to fire all the transitions involved in the conflict.

4 Formalization and Implementation of SITPNs

In this section, we formally present the structure of SITPNs, together with their semantics, and present some parts of the corresponding implementation realized using the Coq proof assistant [14]. Due to lack of space, we purposely omit to present the metatheoretical principles underlying the Coq system. The interested reader can refer to [12] for a thorough introduction to the concepts of Coq.

4.1 SITPN Data Structure

Let us introduce some notations that we use to formalize SITPNs. Let $\mathbb{I}^+$ be the set of intervals of positive integers, where $[a, b] \in \mathbb{I}^+$, and $a \in \mathbb{N}^*$ and $b \in \mathbb{N}^* \sqcup \{\infty\}$. Let us also add that given an interval $I = [a, b] \in \mathbb{I}^+$ and a positive integer $\theta \in \mathbb{N}^*$, $I - \theta = [a - \theta, b - \theta]$ if $b \neq \infty$ and $I - \theta = [a - \theta, b]$ otherwise. Moreover, $lower(I)$ (resp. $upper(I)$) denotes the lower (resp. upper) bound of interval I.

An SITPN is defined as follows (this definition is mainly based on the similar definition given in [9]):

Definition 1 (SITPN). *A synchronously executed, extended, generalized, interpreted, and time Petri net with priorities is a tuple* $<P, T, pre, test, inhib, post, M_0, \succ, \mathcal{A}, \mathcal{C}, \mathcal{F}, \mathbb{A}, \mathbb{C}, \mathbb{F}, I_s>$*, where we have:*

1. $P = \{P_0, \ldots, P_n\}$*, a set of places.*
2. $T = \{T_0, \ldots, T_m\}$*, a set of transitions.*
3. $pre \in P \to T \to \mathbb{N}$*, the function associating a weight to place-transition edges that are not inhibitor or test edges.* $\forall p \in P$, $\forall t \in T$, $pre(p,t) = 0$ *if no such pre edge exists from place* p *to transition* t*.*
4. $test \in P \to T \to \mathbb{N}$*, the function associating a weight to test edges.* $\forall p \in P$, $\forall t \in T$, $test(p,t) = 0$ *if no test edge exists from place* p *to transition* t*.*
5. $inhib \in P \to T \to \mathbb{N}$*, the function associating a weight to inhibitor edges.* $\forall p \in P$, $\forall t \in T$, $inhib(t,p) = 0$ *if no inhibitor edge exists from place* p *to transition* t*.*
6. $post \in T \to P \to \mathbb{N}$*, the function associating a weight to transition-place edges.* $\forall t \in T$, $\forall p \in P$, $post(t,p) = 0$ *if no such post edge exists from transition* t *to place* p*.*
7. $M_0 \in P \to \mathbb{N}$*, the initial marking of the SITPN.*
8. $\succ$*, the irreflexive, anti-symmetric, and transitive priority relation, which represents the firing priority between transitions.*
9. $\mathcal{A} = \{a_0, \ldots, a_n\}$*, a set of continuous actions.*
10. $\mathcal{C} = \{c_0, \ldots, c_n\}$*, a set of conditions.*
11. $\mathcal{F} = \{f_0, \ldots, f_n\}$*, a set of functions (discrete actions).*
12. $\mathbb{A} \in P \to \mathcal{A} \to \mathbb{B}$*, the function associating actions to places.* $\forall p \in P$, $\forall a \in \mathcal{A}$, $\mathbb{A}(p,a) = 1$*, if* a *is associated to* p*,* $\mathbb{A}(p,a) = 0$ *otherwise.*
13. $\mathbb{F} \in T \to \mathcal{F} \to \mathbb{B}$*, the function associating functions to transitions.* $\mathbb{F}$ *is defined in the same manner as* $\mathbb{A}$*.*
14. $\mathbb{C} \in T \to \mathcal{C} \to \{-1, 0, 1\}$*, the function associating conditions to transitions.* $\forall t \in T$, $\forall c \in \mathcal{C}$, $\mathbb{C}(t,c) = 1$*, if* c *is associated to* t*,* $\mathbb{C}(t,c) = -1$*, if* $\bar{c}$ *is associated to* t*,* $\mathbb{C}(t,c) = 0$ *otherwise.*
15. $I_s \in T \nrightarrow \mathbb{I}^+$*, the partial function associating static time intervals to transitions.* T_i *denotes the definition domain of* I_s*, i.e. the set of time transitions.*

From this definition, SITPNs have been implemented in Coq as a record type, i.e. the `Sitpn` type of List. 1. In this code, the types `Place`, `Trans`, `Action`, `Condition`, and `IFunction` are actually aliases for the type of natural numbers (`nat` in Coq). The fields `places` and `transs` respectively represent the set of places and transitions of the PN. The fields `pre`, `test`, and `inhib` are functions returning the weight of place-transition edges, respectively for regular, test, and inhibitor edges. In the same way, the field `post` deals with output edges. The field `initial_marking` corresponds to the initial marking of the PN.

The `s_intervals` field implements the partial function I_s. `s_intervals` returns an `option` of type `TimeInterval`, which is the type implementing the $\mathbb{I}^+$ set. The `option` type allows us to implement partial functions in Coq, where all functions must be total (the `None`/`Some` constructors are used to represent the elements that are respectively members and not members of the domain). The fields `conditions`, `actions`, and `functions` respectively implement the set of

```
Record Sitpn : Set := mk_Sitpn {
    places : list Place;
    transs : list Trans;
    pre : Place → Trans → nat;
    test : Place → Trans → nat;
    inhib : Place → Trans → nat;
    post : Trans → Place → nat;
    initial_marking : Place → nat;
    priority_groups : list (list Trans);
    s_intervals : Trans → option TimeInterval;
    conditions : list Condition;
    actions : list Action;
    functions : list IFunction;
    has_condition : Trans → Condition → bool;
    has_action : Place → Action → bool;
    has_function : Trans → IFunction → bool;
}.
```

Listing 1: The Coq Structure for SITPNs

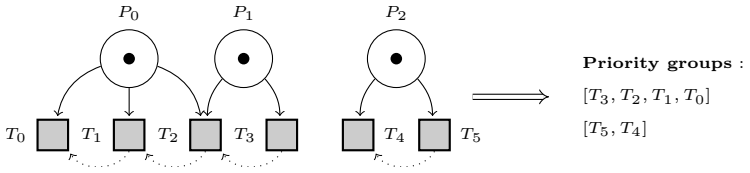


Fig. 5: Computation of Priority Groups of a Petri Net

conditions, actions, and functions. The field `has_condition` implements the $\mathbb{C}$ function, and indicates if a given condition is attached to a given transition. The fields `has_function` and `has_action` similarly implement the $\mathbb{F}$ and $\mathbb{A}$ functions.

As an alternative to the priority relation $\succ$ introduced in Def. 1, the priorities between transitions are implemented by the `priority_groups` field, which contains a list of lists of transitions ordered by level of firing priority. Fig. 5 shows how priority groups are computed from the structure of the PN and the definition of the priority relation (represented by the dotted arrows). To build priority groups, we have to build conflict groups first by gathering all the transitions that have an input place in common. If the intersection of two conflict groups is not empty, then the two conflict groups are merged into one. For example, in Fig. 5, the conflict groups $\{T_0, T_1, T_2\}$ and $\{T_2, T_3\}$ have to be merged. When all the conflict groups have been identified and built, an ordering of transitions that conforms with the definition of the priority relation is applied to each conflict group. This allows us to determine the firing order and create priority groups. If the priority relation is only partially defined for a given conflict group, then the translation into a priority group is not possible. In the following, we only consider models for which the priority relation defines a total order over the transitions of each conflict group.

```
Record SitpnState : Set := mk_SitpnState {
    fired : list Trans;
    marking : list (Place * nat);
    d_intervals : list (Trans * DynamicTimeInterval);
    reset : list (Trans * bool);
    exec_a : list (Action * bool);
    exec_f : list (IFunction * bool);
    cond_values : list (Condition * bool);
}.
```

Listing 2: The Coq Structure for SITPN state

4.2 SITPN Semantics

Before describing the semantics of SITPNs, we need to introduce some preliminary definitions and notations. In the rest of the section, each definition refers to an implicit SITPN $<P, T, pre, test, inhib, post, M_0, \succ, \mathcal{A}, \mathcal{C}, \mathcal{F}, \mathbb{A}, \mathbb{C}, \mathbb{F}, I_s>$.

An SITPN state is defined as follows:

Definition 2 (SITPN State). *An SITPN state is a tuple* $<Fired, M, I, reset_t, ex, cond>$*, where:*

1. $Fired \subseteq T$ *is a set of transitions.*
2. $M \in P \to \mathbb{N}$ *is the current marking of the SITPN.*
3. $I \in T_i \to \mathbb{I}^+ \sqcup \{\psi\}$ *is the function mapping time transitions to dynamic time intervals. A dynamic time interval is either active, then it is in* $\mathbb{I}^+$*, or blocked, then its value is* ψ*. Function* I *denotes the current value of time intervals in the SITPN.*
4. $reset_t \in T_i \to \mathbb{B}$ *is the function mapping time transitions to time interval reset orders (defined as Booleans).*
5. $ex \in \mathcal{A} \sqcup \mathcal{F} \to \mathbb{B}$ *is the function representing the current activation (resp. execution) state of actions (resp. functions).*
6. $cond \in \mathcal{C} \to \mathbb{B}$ *is the function representing the current value of conditions (defined as Booleans).*

As SITPNs, we have implemented the SITPN state structure in Coq as a record type, i.e. the `SitpnState` type of List. 2. Most of the fields of this type are functions, and for easy handling purposes, they are represented as lists of couples. In List. 2, the `DynamicTimeInterval` type implements the set $\mathbb{I}^+ \sqcup \{\psi\}$.

In addition, we introduce the notions of sensitization and firability:

Definition 3 (Sensitization). *A transition* $t \in T$ *is said to be sensitized by a marking* M*, which is noted* $t \in sens(M)$*, if and only if* $\forall p \in P,\ M(p) \geq pre(p,t)$*, and* $\forall p \in P,\ M(p) \geq test(p,t)$*, and* $\forall p \in P,\ M(p) < inhib(p,t)$ *or* $inhib(t) = 0$*.*

Definition 4 (Firability). *A transition* $t \in T$ *is said to be firable at a state* $s = <Fired, M, I, reset_t, ex, cond>$*, which is noted* $t \in firable(s)$*, if and only if* $t \in sens(M)$*, and* $t \notin T_i$ *or* $0 \in I(t)$*, and* $\forall c \in \mathcal{C}, \mathbb{C}(t,c) = 1 \Rightarrow cond(c) = 1$ *and* $\mathbb{C}(t,c) = -1 \Rightarrow cond(c) = 0$*.*

In the following, we will use a simplified notation for markings: given two markings M and M', the formula $\mathcal{O}(M, M')$ will be considered as equivalent to the formula $\forall p \in P,\ \mathcal{O}(M(p), M'(p))$, where $\mathcal{O}$ is a binary relation between two markings of a place.

The semantics of SITPNs is given by the following definition:

Definition 5 (SITPN Semantics). *The semantics of an SITPN is the transition system* $< S, s_0, L, env, \rightsquigarrow >$ *where:*

- S *is the set of states of the SITPN.*
- $s_0 =< \emptyset, M_0, I_s, \emptyset, \emptyset >$ *is the initial state of the SITPN, where* M_0 *is the initial marking of the SITPN, and* I_s *the partial function associating static time intervals to transitions.*
- $L \subseteq Clk \times \mathbb{N}$ *is the set of transition labels, where* $Clk \in \{\uparrow, \downarrow\}$*. A label is a couple* (clk, τ) *composed of a clock event* $clk \in Clk$*, and a time value* $\tau \in \mathbb{N}$ *expressing the current count of clock cycles.*
- $env \in \mathcal{C} \to \mathbb{N} \to \mathbb{B}$ *is the environment function, which gives (Boolean) values to conditions* ($\mathcal{C}$) *depending on the count of clock cycles* ($\mathbb{N}$)*.*
- $\rightsquigarrow \subseteq S \times L \times S$ *is the state transition relation, which is noted* $s \overset{l}{\rightsquigarrow} s'$ *where* $s, s' \in S$ *and* $l \in L$*, and which is defined as follows:*
  - $\forall \tau \in \mathbb{N},\ s \overset{(\downarrow,\tau)}{\rightsquigarrow} s'$*, where* $s =< Fired, M, I, reset_t, ex, cond >$ *and* $s' =< Fired', M, I', reset_t, ex', cond' >$*, if:*
    - *(1)* $cond'$ *is the function giving the (Boolean) values of conditions that are extracted from the environment at the clock count* τ*, i.e.:* $\forall c \in \mathcal{C},\ cond'(c) = env(c, \tau)$.
    - *(2) All the actions associated with at least one marked place in the marking* M *are activated, i.e.:* $\forall a \in \mathcal{A},\ \big(\exists p \in P,\ M(p) > 0 \wedge \mathbb{A}(p, a) = 1\big) \Rightarrow ex'(a) = 1$.
    - *(3) All the actions that are only associated with unmarked places in the marking* M *are deactivated, i.e.:* $\forall a \in \mathcal{A},\ \big(\forall p \in P,\ M(p) = 0 \vee \mathbb{A}(p, a) = 0\big) \Rightarrow ex'(a) = 0$.
    - *(4) All the time transitions that are sensitized by the marking* M*, and received the order to reset their time intervals or were fired at the previous clock cycle, have their dynamic time intervals reset and decremented*[3]*, i.e.:* $\forall t \in T_i,\ t \in sens(M) \wedge \big(reset_t(t) = 1 \vee t \in Fired\big) \Rightarrow I'(t) = I_s(t) - 1$.
    - *(5) All the time transitions with active dynamic time intervals that are sensitized by the marking* M*, did not receive the order to reset their dynamic time intervals, and were not fired at the previous clock cycle, have their dynamic time intervals decremented, i.e.:* $\forall t \in T_i,\ t \in sens(M) \wedge reset_t(t) = 0 \wedge t \notin Fired \wedge I(t) \neq \psi \Rightarrow I'(t) = I(t) - 1$.

[3] Note that decrementing dynamic time intervals allows us to manage local times (by "local", we mean local to a given transition).

(6) *All the time transitions verifying the same conditions as above, but with blocked dynamic time intervals, keep their dynamic time intervals blocked, i.e.:*
$\forall t \in T_i,\ t \in sens(M) \wedge reset_t(t) = 0 \wedge t \notin Fired \wedge I(t) = \psi \Rightarrow I'(t) = I(t)$.

(7) *All the time transitions that are not sensitized by the marking M have their dynamic time intervals reset and decremented, i.e.:*
$\forall t \in T_i,\ t \notin sens(M) \Rightarrow I'(t) = I_s(t) - 1$.

(8) *All the transitions both firable and still sensitized by the residual marking, which is the marking resulting from the firing of all higher priority transitions, are fired, i.e.:*
$\forall t \in firable(s'),\ t \in sens\big(M - \sum_{t_i \in Pr(t)} pre(t_i)\big) \Rightarrow t \in Fired'$, *where* $Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in firable(s')\}$.

(9) *All the firable transitions that are not sensitized by the residual marking are not fired, i.e.:*
$\forall t \in firable(s'),\ t \notin sens\big(M - \sum_{t_i \in Pr(t)} pre(t_i)\big) \Rightarrow t \notin Fired'$.

(10) *All the transitions that are not firable are not fired, i.e.:*
$\forall t \in T,\ t \notin firable(s') \Rightarrow t \notin Fired'$.

- $\forall \tau \in \mathbb{N},\ s' \overset{(\uparrow,\tau)}{\rightsquigarrow} s''$, *where* $s' =< Fired', M', I', reset'_t, ex', cond' >$ *and* $s'' =< Fired', M'', I'', reset''_t, ex'', cond' >$, *if:*

(11) *M'' is the new marking resulting from the firing of all the transitions contained in $Fired'$, i.e.:*
$M'' = M' - \sum_{t_i \in Fired'} \big(pre(t_i) - post(t_i)\big)$.

(12) *All the time transitions that are not sensitized by the transient marking, which is the marking resulting from the retrieval of tokens from the input places of fired transitions, receive the order to reset their dynamic time intervals, i.e.:*
$\forall t \in T_i,\ t \notin sens\big(M' - \sum_{t_i \in Fired'} pre(t_i)\big) \Rightarrow reset''(t) = 1$.

(13) *All the time transitions that are sensitized by the transient marking receive no reset order, i.e.:*
$\forall t \in T_i,\ t \in sens\big(M' - \sum_{t_i \in Fired'} pre(t_i)\big) \Rightarrow reset''(t) = 0$.

(14) *Dynamic time intervals are blocked for all time transitions that have reached the upper bound of their dynamic time intervals and were not fired at this clock cycle, i.e.:*
$\forall t \in T_i,\ upper(I'(t)) = 0 \wedge t \notin Fired' \Rightarrow I''(t) = \psi$.

(15) *Dynamic time intervals are unchanged for all the time transitions that have not reached the upper bound of their dynamic time intervals or were fired at this clock cycle, i.e.:*
$\forall t \in T_i,\ upper(I'(t)) \neq 0 \vee t \in Fired' \Rightarrow I''(t) = I'(t)$.

(16) *All the functions associated with fired transitions are executed, i.e.:*
$\forall f \in \mathcal{F},\ \big(\exists t \in Fired',\ \mathbb{F}(t, f) = 1\big) \Rightarrow ex''(f) = 1$.

(17) *All the functions associated with none of the fired transitions are not executed, i.e.:*
$\forall f \in \mathcal{F},\ \big(\forall t \in Fired',\ \mathbb{F}(t, f) = 0\big) \Rightarrow ex''(f) = 0$.

The SITPN semantics describes how an SITPN goes from one state to another at each occurrence of a falling or rising edge of the clock signal. The clock signal is always alternating consecutively between falling and rising edges. A clock cycle is determined by the period between two falling edges or two rising edges. Also, the count of clock cycles τ is incremented at each clock cycle.

In Def. 5, Rules (1), (2), and (3) deal with interpretation. Rules (4), (5), (6), and (7) deal with dynamic time intervals and their update on falling edge. On falling edge, the dynamic time interval decrementing represents the beginning of a new clock cycle. Rules (8), (9), and (10) deal with the computation of transitions to be fired at the next rising edge. In Rules (8) and (9), the marking $M - \sum_{t_i \in Pr(t)} pre(t_i)$ is the residual marking obtained when all higher priority transitions of a given transition t have consumed tokens in their input places. If a transition t is sensitized by such a residual marking, it means that even after the firing of all its higher priority transitions, enough tokens are left for the firing of t at the same clock event.

Regarding the rising edge rules, Rule (11) computes the marking resulting from the firing of transitions of the $Fired'$ set, which has been determined at the previous clock event. Rules (12) and (13) describe how reset orders are determined based on the sensitization of transitions by the transient marking. The transient marking, represented by the expression $M' - \sum_{t_i \in Fired'} pre(t_i)$, results from the retrieval of tokens from the input places of transitions of the $Fired'$ set. If a transition is disabled by the transient marking, then its dynamic time interval must be reset at the next clock cycle. Rules (14) and (15) determine which dynamic time intervals have reached their upper bounds and became blocked. Rules (16) and (17) deal with the execution of functions. Functions are executed when the corresponding transitions are fired[4].

We implement the SITPN semantics in `Coq` as an inductive relation, whose we show an excerpt in List. 3. The `SitpnSemantics` type describes the behavior of the state transition relation. It consists of two cases parameterized by an instance of the `Clock` type, which has also two constructors `falling_edge` and `rising_edge`. The parameters `tau` and `Clock` represent the label of the state transition relation. The `env` function, which provides values to the conditions at `tau`, is also a parameter of the relation.

Due to lack of space, we only provide the code of some rules for the second case of the SITPN semantics, i.e. some rules of state evolution on rising edge. More precisely, we give the code of Rules (11), (12), (14), and (16). Note that to access the fields of record types, such as `Sitpn` and `SitpnState`, we use the prefix notation. For example, given `s` of type `SitpnState`, we write `(marking s)` and `(fired s)` to respectively access the `marking` and `fired` fields of `s`.

The implementation of Rule (11) expresses the computation of the marking M' at the state s', i.e. `(marking s')`, according to the marking M and the list of fired transitions at the state s, i.e. `(marking s)` and `(fired s)` respectively. As markings are lists of couples (each couple consists of a place and its

[4] The execution of actions and functions affects the environment of the SITPN, which gives feedbacks through the values provided to the conditions.

```
Inductive SitpnSemantics (sitpn : Sitpn) (s s' : SitpnState) (tau : nat)
          (env : Condition → nat → bool) : Clock → Prop :=
| SitpnSemantics_falling_edge : (* Rules 1 to 10 *)
  ...  → SitpnSemantics sitpn s s' tau env falling_edge
| SitpnSemantics_rising_edge :
  (** Premises **)
  (* Ensures the consistency of sitpn, s and s'. *)
  IsWellDefinedSitpn sitpn →
  IsWellDefinedSitpnState sitpn s →
  IsWellDefinedSitpnState sitpn s' →
  (* Rule 11 *)
  (forall (p : Place) (n : nat),
    In (p, n) (marking s) →
    In (p, n − (pre_sum sitpn p (fired s)) + (post_sum sitpn p (fired s)))
    (marking s')) →
  (* Rule 12 *)
  (forall (t : Trans) (transient_marking : list (Place * nat)),
    Permutation (places sitpn) (fs transient_marking) →
    (forall (p : Place) (n : nat),
      In (p, n) (marking s) →
      In (p, n − pre_sum sitpn p (fired s)) transient_marking) →
    In t (transs sitpn) →
    s_intervals sitpn t <> None →
    ~IsSensitized sitpn transient_marking t →
    In (t, true) (reset s')) →
  (* Rule 14 *)
  (forall (t : Trans) (dyn_itval : DynamicTimeInterval),
    In (t, dyn_itval) (d_intervals s) →
    HasReachedUpperBound dyn_itval →
    ~In t (fired s) →
    In (t, blocked) (d_intervals s')) →
  (* Rule 16 *)
  (forall (f : IFunction),
    In f (functions sitpn) →
   (exists (t : Trans), In t (fired s) ∧ (has_function sitpn t f) = true) →
    In (f, true) (exec_f s')) →
  (** Conclusion **)
  SitpnSemantics sitpn s s' tau env rising_edge.
```

Listing 3: The Semantics of SITPNs in Coq

number of tokens), contents of markings are expressed by the mean of the `In` relation[5]. The `pre_sum` and `post_sum` functions implement the sum expression $\sum_{t_i \in Fired} \big(pre(t_i) - post(t_i)\big)$ used in Rule (11). Regarding Rule (12), we express the transient marking in the same way as the marking M' in Rule (11), i.e. with the `In` relation and the `pre_sum` function. The `Permutation` relation ensures the well-definition of the list `transient_marking`, by checking that all the places of the SITPN are referenced in the transient marking. In the implementation of Rule (14), the line `(In (t,dyn_itval) (d_intervals s))` implicitly implies that $t \in T_i$. This is a consequence of the well-definition of object `s`, which is asserted by the premise `(IsWellDefinedSitpnState sitpn s)`.

[5] `(In a l)` means that the element `a` is in the list `l`.

When defining instances of `Sitpn` and `SitpnState`, because of the permissiveness of the data structures used in our implementation (mostly lists), there may be some inconsistencies (places not referenced in markings, transitions with no edges, etc.). We therefore define the predicate `IsWellDefinedSitpn`, as well as the relation `IsWellDefinedSitpnState`, to prevent these inconsistencies, and to provide a safe basis for proofs. Also, these inconsistencies are the causes of the errors that may be returned by our token player (see List. 4), which may occur when computing the state of a given SITPN (if this SITPN is not well-defined or if the computation is performed from a not well-defined state).

5 A Verified Token Player for SITPNs

In this section, we describe the algorithm of a token player for SITPNs, along with its implementation in Coq. We then outline the soundness and completeness proofs of this token player with respect to the SITPN semantics. Building a token player that follows the formalized and mechanized SITPN semantics is a way to assess our implementation of the semantics. Moreover, it also provides the users of the SITPN formalism with a verified and therefore reliable token player.

5.1 A Token Player for SITPNs

An token player for SITPNs is a program taking an SITPN and its initial state as parameters and returning an ordered list of SITPN states corresponding to the evolution of the SITPN over n clock cycles. Before describing the Coq implementation of such a token player, we first present the algorithms (using imperative pseudo-code) that allows us to compute of a new state for an SITPN after one clock cycle. Algo. 1 computes the SITPN state after the falling edge of the clock, while Algo. 2 computes the SITPN state after the rising edge of the clock. Due to lack of space, we only provide, in List. 4, the Coq implementation of Algo. 2. In Algos. 1 and 2, the variables for SITPNs and SITPN states are records respectively of type `Sitpn` and `SitpnState` (see Lists. 1 and 2). The assignment of values to variables and record fields is performed with the "$\leftarrow$" operator.

Algo. 1 computes of a new state s' for *sitpn* starting from state s with respect to the falling edge semantics defined in Def. 5. At the beginning of the algorithm, the new state s' is initialized with state s (Line 2). The fields of state s' are then overrided by the computation of new condition, action state, time interval values and a new list of transitions to be fired.

The function `getConditionValues` (Line 3) provides a list of couples (c, v), where c is a condition of the SITPN, and v its current (Boolean) value at time t, computed by the *env* function. The function `getConditionValues` implements Rule (1) of the SITPN semantics.

The function `getActionStates` (Line 4) returns a list of couples (a, v), where a is an action of the SITPN, and v its current Boolean-valued activation state. For each action, the function tries to find a place with a marking greater than zero with which this action is associated. If such a place exists then the action

Algorithm 1: falling_edge(*sitpn*, *s*, *t*, *env*)

Data: *sitpn*, an SITPN. *s*, the state of *sitpn* before the falling edge of the clock. *t*, the current count of clock cycles. *env*, the function that provides condition values at each clock cycle.
Result: s', the new state of *sitpn* after the falling edge of the clock.

```
1 begin
2     s' ← s
3     (cond_values s') ← getConditionValues(sitpn, t, env)
4     (exec_a s') ← getActionStates(sitpn, s)
5     (d_intervals s') ← updateTimeIntervals(sitpn, s)
6     (fired s') ← mapFire(sitpn, s')
7     return s'
```

Algorithm 2: rising_edge(*sitpn*, *s*)

Data: *sitpn*, an SITPN. *s*, the state of *sitpn* before the rising edge of the clock.
Result: s', the new state of *sitpn* after the rising edge of the clock.

```
1 begin
2     s' ← s
3     transient_marking ← mapUpdateMarkingPre(sitpn, s)
4     (reset', d_intervals') ← getBlItvalsAndResetOrds(sitpn, s, transient_marking)
5     (reset s') ← reset'
6     (d_intervals s') ← d_intervals'
7     (marking s') ← mapUpdateMarkingPost(sitpn, s')
8     (exec_f s') ← getFunctionStates(sitpn, s')
9     return s'
```

is marked as activated. The function `getActionStates` implements Rules (2) and (3) of the SITPN semantics.

The function `updateTimeIntervals` (Line 5) implements Rules (4), (5), (6) and (7) of the SITPN semantics. The function iterates over the elements of the list `(d_intervals s)`, which are couples (t, ti), where t is a time transition and ti is its associated dynamic time intervals, and updates the value of each dynamic time interval according to the current state of the corresponding transition.

The function `mapFire` computes the list of transitions to be fired at this cycle of evolution. In the body of the function, all the transitions are checked, proceeding priority group by priority group. A residual marking, as described in Rules (8) and (9) of the semantics, is computed for each priority group. The residual marking contributes to the selection of transitions to be fired. If a transition *trans*, in a given priority group, is firable and sensitized by the residual marking, then it is selected as a transition to be fired, and a new residual marking is computed by withdrawing the appropriate number of tokens from the input places of *trans*.

For a given *sitpn* at a current state *s*, Algo. 2 computes a new state s' with respect to the evolution semantics defined for the rising edge event of a clock signal. The function `mapUpdateMarkingPre` (Line 3) implements the first

```
1  Definition sitpn_rising_edge (sitpn : Sitpn) (s : SitpnState) :
2        option SitpnState :=
3    match map_update_marking_pre sitpn (marking s) (fired s) with
4    | Some transient_marking ⇒
5      match get_blocked_itvals_and_reset_orders sitpn s (d_intervals s)
6            transient_marking [] [] with
7      | Some (reset', d_intervals') ⇒
8        match map_update_marking_post sitpn transient_marking (fired s) with
9        | Some final_marking ⇒
10         let exec_f' := get_function_states sitpn s (functions sitpn) [] in
11         Some (mk_SitpnState (fired s) final_marking d_intervals' reset'
12                             (cond_values s) (exec_a s) exec_f')
13       (* Error raised by map_update_marking_post. *)
14       | None ⇒ None
15       end
16     (* Error raised by get_blocked_itvals_and_reset_orders. *)
17     | None ⇒ None
18     end
19   (* Error raised by map_update_marking_pre. *)
20   | None ⇒ None
21   end.
```

Listing 4: The Implementation of Algo. 2 in Coq

part of the firing process and returns a transient marking, result of the retrieval of tokens from the input places of the transitions of the `(fired` *sitpn*`)` list, i.e. the transitions to be fired at this clock cycle. The firing process is not done all at once as we need the transient marking to determine which dynamic time intervals must be reset at the next clock cycle. This is done by the function `getBlItvalsAndResetOrds` (Line 4), which both assigns reset orders for transitions either disabled by the transient marking or fired at this clock cycle, and determines which dynamic time intervals are blocked. This function returns a couple (ro, ti), where ro is a list of reset orders and ti is a list of dynamic time intervals, and the returned values ro and ti are then assigned to the corresponding fields of the new `SitpnState` instance s' (Lines 5 and 6). This function implements Rules (12), (13), (14), and (15) of the SITPN semantics. The function `mapUpdateMarkingPost` (Line 7) produces new tokens in the output places of transitions that are in the list `(fired` *sitpn*`)`, which is the end of the firing process. The functions `mapUpdateMarkingPre` and `mapUpdateMarkingPost` implements Rule (11) of the SITPN semantics. Finally, the function `getFunctionStates` (Line 8) computes the state of functions based on the list of fired transitions `(fired` *sitpn*`)`. This function implements Rule (16) and (17) of the SITPN semantics.

List. 4 provides the Coq implementation of Algo. 2, of which it is almost a strict transcription. Note that the functions `map_update_marking_pre`/`post`, and `get_blocked_itvals_and_reset_orders` may produce errors (i.e., the constructor `None` of the type `option`).

The main function is the function `sitpn_cycle`, which is not presented here, and which executes in sequence the functions `sitpn_falling_edge` and `sitpn_rising_edge`. These two functions are respective implementations of Algos. 1 and 2. The function `sitpn_cycle` computes a complete evolution cycle for a given `Sitpn` structure, and computes a couple of states corresponding to the states of the SITPN after the falling and rising edges of the clock signal.

5.2 Soundness and Completeness of our Token Player

To improve the confidence in our implementation of the SITPN semantics, which will be used later in the proof that the HILECOP methodology preserves the behavior of its models (see Sec. 1), the soundness and completeness of the function `sitpn_cycle` with respect to the SITPN semantics have been proved. Only an overview of the proof is given here, and the interested reader can access to the full source code[6] to get all the details of the formalization.

Proof of Soundness The following theorem states the soundness of the function `sitpn_cycle`, i.e. the two states `s'` and `s''` returned by the function comply with the SITPN semantics:

Theorem 1. $\forall$ *sitpn* $\in SITPN$, $\forall s, s', s'' \in S$, $\forall \tau \in \mathbb{N}$, $\forall \texttt{env} \in \mathcal{C} \to \mathbb{N} \to \mathbb{B}$, *(sitpn_cycle sitpn s env* τ*) = Some (s', s'')* $\Rightarrow s \overset{(\downarrow,\tau)}{\rightsquigarrow} s' \overset{(\uparrow,\tau)}{\rightsquigarrow} s''$.

As the function `sitpn_cycle` is divided into two parts, i.e. the call to the functions `sitpn_falling_edge` and `sitpn_rising_edge`, the proof is divided into two parts as well, with two intermediary lemmas. The first lemma states that the function `sitpn_falling_edge` returns a SITPN state that conforms with the falling edge semantics:

Lemma 1. $\forall$ *sitpn* $\in SITPN$, $\forall s, s' \in S$, $\forall \tau \in \mathbb{N}$, $\forall \texttt{env} \in \mathcal{C} \to \mathbb{N} \to \mathbb{B}$, *(sitpn_falling_edge sitpn s env* τ*) = Some s'* $\Rightarrow s \overset{(\downarrow,\tau)}{\rightsquigarrow} s'$.

Proving this lemma is equivalent to proving that the functions called in the function `sitpn_falling_edge`, which implement the functions of Algo. 1, imply a part of the rules of the SITPN semantics. Auxiliary lemmas have therefore been proved for each of these functions:

- The function `getConditionValues` implies Rule (1);
- The function `getActionStates` implies Rules (2) and (3);
- The function `updateTimeIntervals` implies Rules (4), (5), (6), and (7);
- The function `mapFire` implies Rules (8), (9), and (10).

The second intermediary lemma states that the function `sitpn_rising_edge` returns a SITPN state that conforms with the rising edge semantics:

[6] The full code is available at:
`https://github.com/viampietro/sitpns`.

Lemma 2. $\forall$ *sitpn* $\in SITPN$, $\forall$ *s, s'* $\in S$, $\forall \tau \in \mathbb{N}$, $\forall$ env $\in \mathcal{C} \to \mathbb{N} \to \mathbb{B}$, *(sitpn_rising_edge sitpn s) = Some s'* $\Rightarrow$ *s* $\overset{(\uparrow,\tau)}{\rightsquigarrow}$ *s'*.

Just as with Lem. 1, the proof consists in proving that the functions used in the function `sitpn_rising_edge` imply some rules of the semantics.

Most of the functions that are used in the functions `sitpn_falling_edge` and `sitpn_rising_edge` iterate over the structure of the fields of the SITPN and SITPN state structures. For instance, the function `updateTimeIntervals` iterates over the list of dynamic time intervals of the SITPN state `s` (see Algo. 1). Proving that a function implies some rules of the semantics is therefore naturally done by structural induction on the field which the function iterates over.

Proof of Completeness The completeness theorem is expressed as follows:

Theorem 2. $\forall$ *sitpn* $\in SITPN$, $\forall$ *s,s', s''* $\in S$, $\forall \tau \in \mathbb{N}$, $\forall$ env $\in \mathcal{C} \to \mathbb{N} \to \mathbb{B}$, *s* $\overset{(\downarrow,\tau)}{\rightsquigarrow}$ *s'* $\overset{(\uparrow,\tau)}{\rightsquigarrow}$ *s''* $\Rightarrow$ *(sitpn_cycle sitpn s env* τ*) = Some (s', s'')*.

The proof of this theorem follows similar patterns as the proof of Thm. 1. In particular, as the function `sitpn_cycle` is partial, we have to prove that, thanks to the hypothesis of well-definition assumed over the parameters `sitpn` and `s`, error cases never occur (i.e. `sitpn_cycle` never returns `None`).

6 Conclusion

In this paper, we have introduced the HILECOP methodology, a process for the design of critical digital systems. Due to the specific context of critical applications, we have proposed to formally verify this methodology, which is motivated by a need for reliability. To do so, we have focused, in this paper, on the formalization of the high level modeling language of HILECOP, which relies in particular on SITPNs to model the behavior of components. This formalization has been implemented using the Coq proof assistant, and we have also provided a token player, which has been proved sound and complete with respect to the semantics of SITPNs. The full implementation is significant and has about 20,000 lines of code, including 250 lemmas and 18,000 lines of proof.

As perspectives, we need to complete the formalization and implementation of the high level modeling language of HILECOP. In particular, we have to deal with macroplaces, which are a way to handle exceptions in a PN [8] and in HILECOP models in particular, and which remain to be integrated into our implementation.

The process of formally verifying the entire HILECOP transformation chain is a long way to go. The next step is to deal with the target language, i.e. VHDL, where the idea is to give a synchronous operational semantics to VHDL, based on previous work of formalization [7]. The implementation of this semantics in Coq is a work in progress.

The final step of our work will be to implement the model-to-text transformation from HILECOP PN models to VHDL code, and to prove that this transformation preserves the semantics of the input models.

References

1. J.-R. Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.
2. K. Berramla, E. A. Deba, and M. Senouci. Formal Validation of Model transformation with Coq Proof Assistant. In *New Technologies of Information and Communication (NTIC)*, pages 1–6, Nov. 2015.
3. T. Bourke, L. Brun, P. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A Formally Verified Compiler for Lustre. In *Programming Language Design and Implementation (PLDI)*, pages 586–601, Barcelona (Spain), June 2017. ACM.
4. D. Calegari, C. Luna, N. Szasz, and A. Tasistro. A Type-Theoretic Framework for Certified Model Transformations. In *Brazilian Symposium on Formal Methods (SBMF)*, volume 6527 of *LNCS*, pages 112–127, Natal (Brazil), Nov. 2010. Springer.
5. C. Choppy, M. Mayero, and L. Petrucci. Experimenting Formal Proofs of Petri Nets Refinements. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 214:231–254, June 2008.
6. L. Fronc and F. Pommereau. Towards a Certified Petri Net Model-Checker. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 7078 of *LNCS*, pages 322–336, Kenting (Taiwan), Dec. 2011. Springer.
7. C. D. Kloos and P. Breuer. *Formal Semantics for VHDL*. Springer, Feb. 1995. ISBN 0792395522.
8. H. Leroux. Handling Exceptions in Petri Net-Based Digital Architecture: From Formalism to Implementation on FPGAs. *IEEE Transactions Industrial Informatics*, 11(4):897–906, Aug. 2015.
9. H. Leroux, K. Godary-Dejean, and D. Andreu. Complex Digital System Design: A Methodology and Its Application to Medical Implants. In *Formal Methods for Industrial Critical Systems (FMICS)*, pages 94–107, Madrid (Spain), Sept. 2013.
10. X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM (CACM)*, 52(7):107–115, July 2009.
11. S. Meghzili, A. Chaoui, M. Strecker, and E. Kerkouche. On the Verification of UML State Machine Diagrams to Colored Petri Nets Transformation Using Isabelle/HOL. In *Information Reuse and Integration (IRI)*, pages 419–426, San Diego (CA, USA), Aug. 2017. IEEE Computer Society.
12. C. Paulin-Mohring. Introduction to the Coq Proof-Assistant for Practical Software Verification. In *LASER Summer School on Software Engineering*, volume 7682, pages 45–95, Elba Island (Italy), Sept. 2011. Springer.
13. Y. K. Tan, M. O. Myreen, R. Kumar, A. C. J. Fox, S. Owens, and M. Norrish. A New Verified Compiler Backend for CakeML. In *International Conference on Functional Programming (ICFP)*, pages 60–73, Nara (Japan), Sept. 2016. ACM.
14. The Coq Development Team. *Coq, version 8.11.0*. Inria, Jan. 2020. `http://coq.inria.fr/`.
15. The IEEE Organization. IEEE Standard VHDL Language Reference Manual – Redline. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) – Redline*, pages 1–620, Jan. 2009.
16. Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin. From AADL to Timed Abstract State Machines: A Verified Model Transformation. *Journal of Systems and Software (JSS)*, 93:42–68, July 2014.