Designing Critical Digital Systems.

Formal Verification of a Token Player for Synchronously Executed Petri Nets.

PhD student:
Vincent Iampietro[1]

PhD supervisors:
David Andreu[1,2], David Delahaye[1]

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

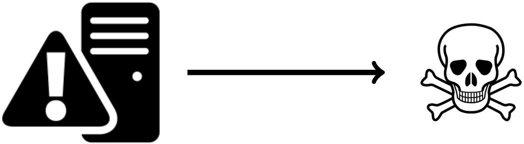
[2]NEURINNOV, Montpellier, France
`David.Andreu@neurinnov.com`

SHARC, July 2019

Context.

CRITICAL DIGITAL SYSTEMS (CDS)?

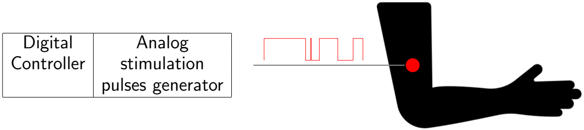
CRITICAL DIGITAL SYSTEMS (CDS)



CRITICAL DIGITAL SYSTEMS (CDS)

- Avionics: engine control, air traffic control. . .
- Medicine: surgical robots, radiotherapy systems. . .
- Spaceflight: launcher systems, crew transfer systems. . .
- Nuclear: reactor control systems. . .
- Infrastructure: fire alarm, telecommunications. . .
- And many more. . .

Medical Implants: A Concrete Example of CDS.



- Electrode receives electric current from stimulation generator.
- Digital controller gives instruction to stimulation generator.

Need for Safety and Certification.

CE Certification for Medical Devices. [1]

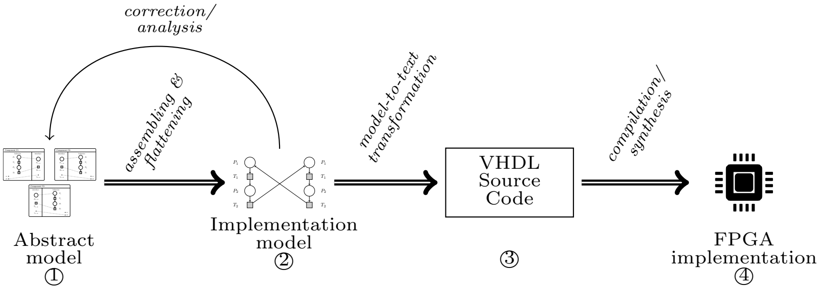
- European Regulation on Medical Devices (2017/745).
- Requires numerous tests on devices (technologic, clinical).

The Perks of Formal Methods.

- Many approaches: model checking, abstract interpretation, deductive methods...
- Deductive methods: test exhaustiveness through proofs.

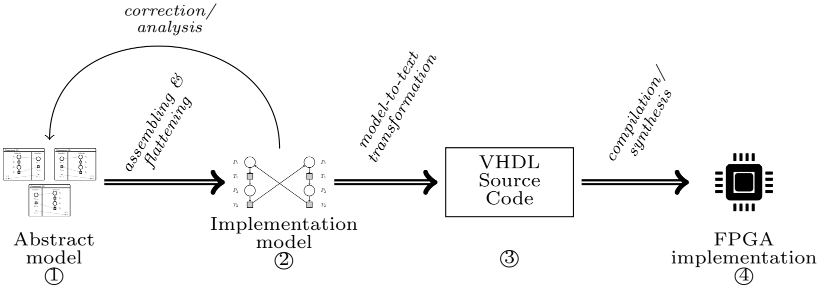
[1] https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32017R0745

HILECOP: A Process to Design and Implement CDS.



- Developed at INRIA (CAMIN Team).

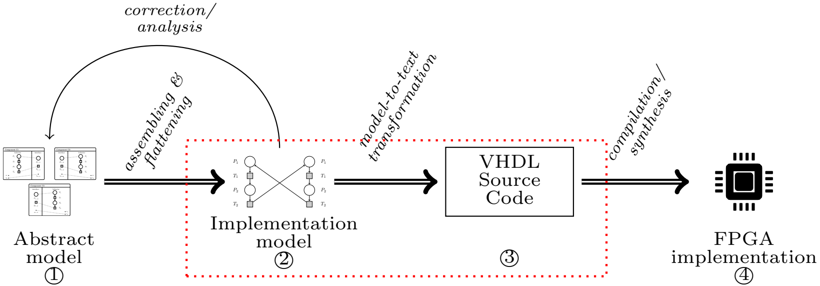
Formal Methods for HILECOP.



Verification of HILECOP.

- Ensure model correctness (analysis).
- Ensure behavior preservation through transformation.

Formal Methods for HILECOP.



Verification of HILECOP.

- Ensure model correctness (analysis).
- Ensure behavior preservation through transformation.

Deductive Methods for HILECOP.

Deductive Methods with the Coq Proof Assistant.

- General-purpose Programming Language.
- Proof Language.

Deductive Methods for HILECOP.

Deductive Methods with the Coq Proof Assistant.

- General-purpose Programming Language.
- Proof Language.

Proof steps.

Inspired by *CompCert*, a formally verified C compiler:

Deductive Methods for HILECOP.

Deductive Methods with the Coq Proof Assistant.

- General-purpose Programming Language.
- Proof Language.

Proof steps.

Inspired by *CompCert*, a formally verified C compiler:

1. Model the semantics of the *source language* (i.e, Petri nets).

Deductive Methods for HILECOP.

Deductive Methods with the Coq Proof Assistant.

- General-purpose Programming Language.
- Proof Language.

Proof steps.

Inspired by *CompCert*, a formally verified C compiler:

1. Model the semantics of the *source language* (i.e, Petri nets).
2. Model the semantics of the *target language* (i.e, VHDL).

Deductive Methods for HILECOP.

Deductive Methods with the Coq Proof Assistant.

- General-purpose Programming Language.
- Proof Language.

Proof steps.

Inspired by *CompCert*, a formally verified C compiler:

1. Model the semantics of the *source language* (i.e, Petri nets).
2. Model the semantics of the *target language* (i.e, VHDL).
3. Implement the transformation.

Deductive Methods for HILECOP.

Deductive Methods with the Coq Proof Assistant.

- General-purpose Programming Language.
- Proof Language.

Proof steps.

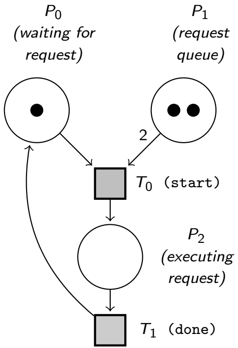
Inspired by *CompCert*, a formally verified C compiler:

1. Model the semantics of the *source language* (i.e, Petri nets).
2. Model the semantics of the *target language* (i.e, VHDL).
3. Implement the transformation.
4. Prove behavior preservation.

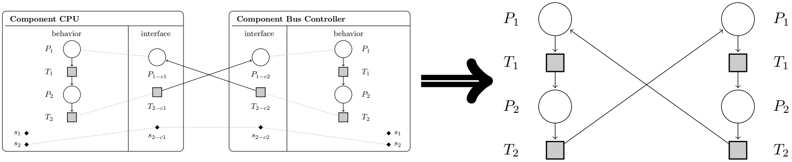
Presentation of HILECOP Petri Nets.

The Petri Net (PN) Formalism.

- To model *dynamic systems*.
- Directed weighted graph.
- Places ($\approx$ states or resources) and transitions ($\approx$ events).
- Marking: current state of the system.
- Sensitization: a transition t is ready to be fired.



HILECOP High-Level Models.

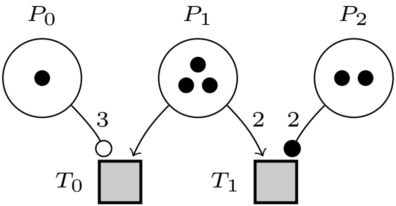


- Assembling components.
- Flattening model.

HILECOP PNs (SITPNs).

HILECOP Petri Nets are:

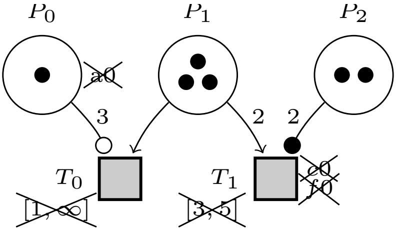
- **S**ynchronously executed (with priorities)
- generalized
- extended
- **I**nterpreted
- **T**ime
- with macroplaces
- **P**etri **N**ets



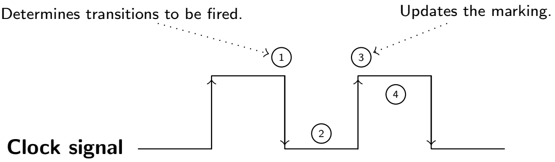
HILECOP PNs (SITPNs).

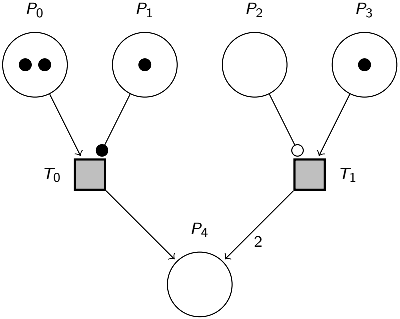
HILECOP Petri Nets are:

- **S**ynchronously executed (with priorities)
- generalized
- extended
- ~~**I**nterpreted~~
- ~~**T**ime~~
- ~~with macroplaces~~
- **P**etri **N**ets

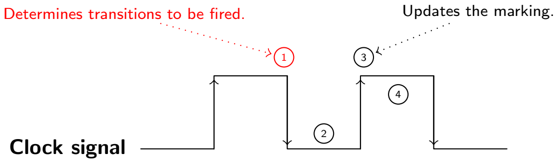


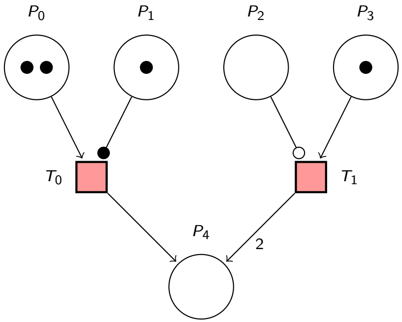
Synchronously Executed PNs.



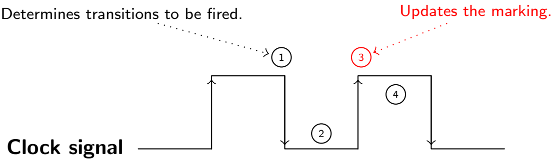


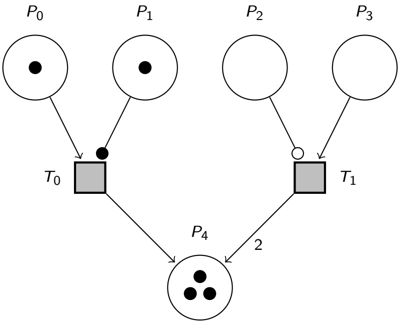
Synchronously Executed PNs.



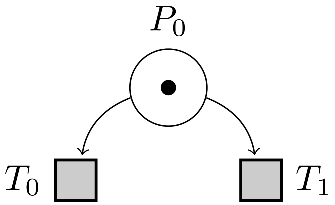


Synchronously Executed PNs.





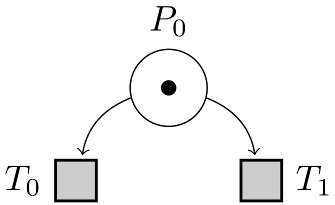
Conflicts and priorities.



Conflict types.

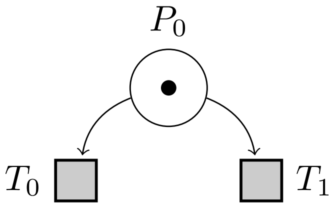
- Structural: T_0 and T_1 have P_0 as a common input place.
- Effective: the firing of T_0 disables T_1, and conversely.

Conflicts and priorities.



Which transition will be fired?

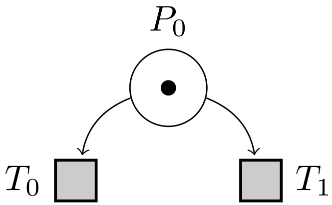
Conflicts and priorities.



Which transition will be fired?

- If asynchronous execution: T_0 or T_1

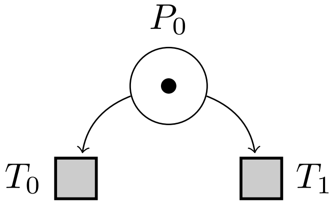
Conflicts and priorities.



Which transition will be fired?

- If asynchronous execution: T_0 or T_1
- If synchronous execution: T_0 and T_1

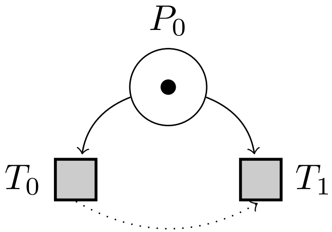
Conflicts and priorities.



Which transition will be fired?

- If asynchronous execution: T_0 or T_1
- If synchronous execution: T_0 and T_1 ⚠

Conflicts and priorities.



Priority relation.

T_0 has a higher firing priority than T_1.

Formalizing HILECOP Petri Nets.

Formal Definition of SPNs.

A synchronously executed, extended, and generalized Petri net with priorities is a tuple $<P, T, pre, test, inhib, post, M_0, clock, \succ>$ where we have:

1. $P = \{P_0, \ldots, P_n\}$ a set of places.
2. $T = \{T_0, \ldots, T_n\}$ a set of transitions.
3. $pre \in P \rightarrow T \rightarrow \mathbb{N}$.
4. $test \in P \rightarrow T \rightarrow \mathbb{N}$.
5. $inhib \in P \rightarrow T \rightarrow \mathbb{N}$.
6. $post \in T \rightarrow P \rightarrow \mathbb{N}$.
7. $M_0 \in P \rightarrow \mathbb{N}$, the initial marking of the SPN.
8. $\succ$, the priority relation, which represents the firing priority between transitions of the same priority group.

Implementation of SPNs in Coq.

```
Structure Spn : Set :=
  mk_Spn {
     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);
     lneighbors : Trans → Neighbors;
  }.
```

- Record with multiple fields.
- `lneighbors` field associates transitions to input/output places.

Definitions and Notations.

Remark.

The following definitions are given under the scope of a SPN $<P, T, pre, test, inhib, post, M_0, \succ>$.

Definitions and Notations.

Remark.

The following definitions are given under the scope of a SPN $<P, T, pre, test, inhib, post, M_0, \succ>$.

Definition (SPN state)

A SPN state is a couple $(Fired, M)$ where $M \in P \rightarrow \mathbb{N}$ is the current marking of SPN and $Fired \subseteq T$ is a list of transitions.

Definitions and Notations.

Remark.

The following definitions are given under the scope of a SPN $<P, T, pre, test, inhib, post, M_0, \succ>$.

Definition (SPN state)

A SPN state is a couple $(Fired, M)$ where $M \in P \to \mathbb{N}$ is the current marking of SPN and $Fired \subseteq T$ is a list of transitions.

Definition (Sensitization and Firability)

- Sensitization: A transition $t \in sens(M)$, if $M \geq pre(t)$, and $M \geq test(t)$, and $M < inhib(t)$ or $inhib(t) = 0$.
- Firability: A transition $t \in firable(s)$, where $s = (Fired, M)$, if $t \in sens(M)$.

SPN Semantics.

Definition (SPN Semantics)

The semantics of an SPN is represented by the triplet $< S, s_0, \rightsquigarrow >$ where:

SPN Semantics.

Definition (SPN Semantics)

The semantics of an SPN is represented by the triplet $< S, s_0, \rightsquigarrow >$ where:

- S is the set of states of the SPN.

SPN Semantics.

Definition (SPN Semantics)

The semantics of an SPN is represented by the triplet $< S, s_0, \rightsquigarrow >$ where:

- S is the set of states of the SPN.
- $s_0 = (\emptyset, M_0)$ is the initial state of the SPN.

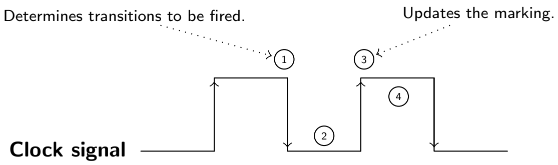
SPN Semantics.

Definition (SPN Semantics)

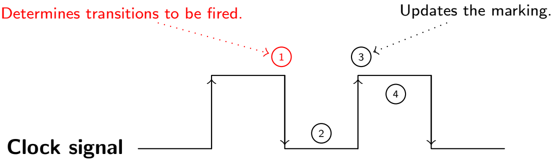
The semantics of an SPN is represented by the triplet $< S, s_0, \rightsquigarrow >$ where:

- S is the set of states of the SPN.
- $s_0 = (\emptyset, M_0)$ is the initial state of the SPN.
- $\rightsquigarrow \subseteq S \times Clk \times S$ is the state changing relation, which is noted $s \overset{clk}{\rightsquigarrow} s'$ where $s, s' \in S$, $Clk = \{\downarrow clock, \uparrow clock\}$ and $clk \in Clk$.

SPN State Changing Relation (Falling Edge).

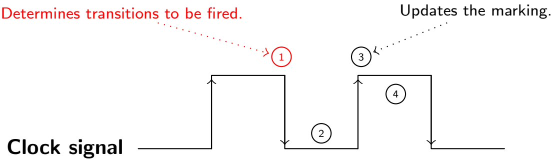


SPN State Changing Relation (Falling Edge).



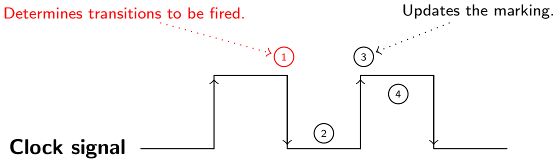
- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if:

SPN State Changing Relation (Falling Edge).



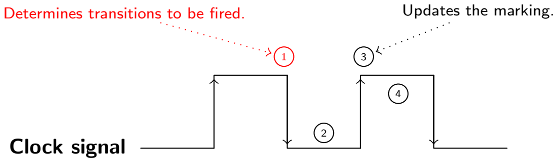
- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if:
  - **All transitions that are not firable are not fired, i.e.:**

SPN State Changing Relation (Falling Edge).



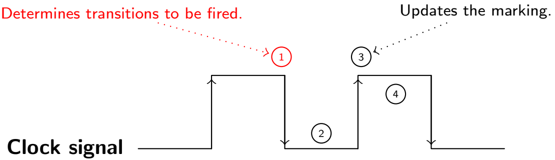
- $s = (Fired, M) \overset{\downarrow clock}{\leadsto} s' = (Fired', M)$ if:
    - **All transitions that are not firable are not fired, i.e.:** $\forall t \in T,\ t \notin firable(s) \Rightarrow t \notin Fired'$.

SPN State Changing Relation (Falling Edge).



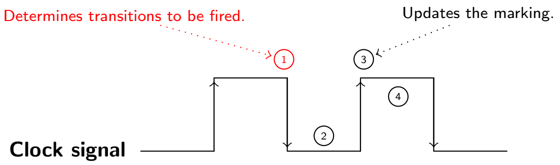
- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if:
  - **All transitions that are not firable are not fired, i.e.:**
    $\forall t \in T,\ t \notin firable(s) \Rightarrow t \notin Fired'$.
  - **All transitions both firable and sensitized by the residual marking are fired, i.e:**

SPN State Changing Relation (Falling Edge).



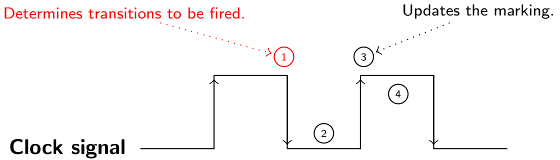
- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if:
  - **All transitions that are not firable are not fired, i.e.:** $\forall t \in T,\ t \notin firable(s) \Rightarrow t \notin Fired'$.
  - **All transitions both firable and sensitized by the residual marking are fired, 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 $Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in Fired'\}$.

SPN State Changing Relation (Falling Edge).



- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if:
  - **All transitions that are not firable are not fired, i.e.:** $\forall t \in T,\ t \notin firable(s) \Rightarrow t \notin Fired'$.
  - **All transitions both firable and sensitized by the residual marking are fired, i.e:** $\forall t \in firable(s),\ t \in sens\left(M - \sum_{t_i \in Pr(t)} pre(t_i)\right) \Rightarrow t \in Fired'$, where $Pr(t) = \{t_i \mid t_i \succ t \land t_i \in Fired'\}$.
  - **All firable transitions that are not sensitized by the residual marking are not fired, i.e.:**

SPN State Changing Relation (Falling Edge).



- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if:
  - **All transitions that are not firable are not fired, i.e.:** $\forall t \in T,\ t \notin firable(s) \Rightarrow t \notin Fired'$.
  - **All transitions both firable and sensitized by the residual marking are fired, i.e:** $\forall t \in firable(s),\ t \in sens\left(M - \sum_{t_i \in Pr(t)} pre(t_i)\right) \Rightarrow t \in Fired'$, where $Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in Fired'\}$.
  - **All firable transitions that are not sensitized by the residual marking are not fired, i.e.:** $\forall t \in firable(s),\ t \notin sens\left(M - \sum_{t_i \in Pr(t)} pre(t_i)\right) \Rightarrow t \notin Fired'$.

An Example of SPN Semantics Rule.

All transitions both firable and sensitized by the residual marking are fired.

$$s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$$

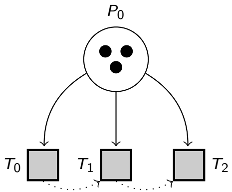


Figure: At state s.

An Example of SPN Semantics Rule.

All transitions both firable and sensitized by the residual marking are fired.

$$s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$$

- $T_0,\ T_1 \in Fired'$

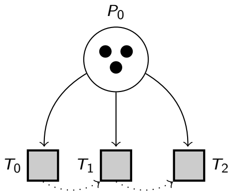


Figure: At state s.

An Example of SPN Semantics Rule.

All transitions both firable and sensitized by the residual marking are fired.

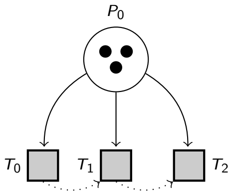


Figure: At state s.

$$s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$$

- $T_0,\ T_1 \in Fired'$
- $T_2 \in Fired'?$

An Example of SPN Semantics Rule.

All transitions both firable and sensitized by the residual marking are fired.

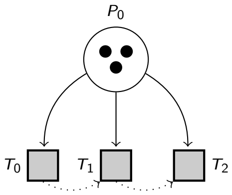


Figure: At state s.

$$s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$$

- $T_0,\ T_1 \in Fired'$
- $T_2 \in Fired'$?
- $M = (P_0, 3),\ T_2 \in firable(s)$?

An Example of SPN Semantics Rule.

All transitions both firable and sensitized by the residual marking are fired.

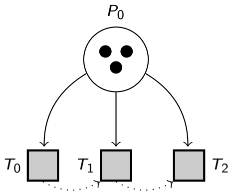


Figure: At state s.

$$s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$$

- $T_0,\ T_1 \in Fired'$
- $T_2 \in Fired'$?
- $M = (P_0, 3),\ T_2 \in firable(s)$? YES!

An Example of SPN Semantics Rule.

All transitions both firable and sensitized by the residual marking are fired.

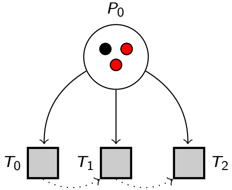


Figure: At state s.

$$s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$$

- $T_0,\ T_1 \in Fired'$
- $T_2 \in Fired'$?
- $M = (P_0, 3),\ T_2 \in firable(s)$? YES!
- $M_R = (P_0, 1),\ T_2 \in sens(M_R)$?

An Example of SPN Semantics Rule.

All transitions both firable and sensitized by the residual marking are fired.

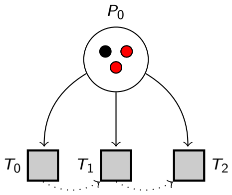


Figure: At state s.

$$s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$$

- $T_0,\ T_1 \in Fired'$
- $T_2 \in Fired'$?
- $M = (P_0, 3),\ T_2 \in firable(s)$? YES!
- $M_R = (P_0, 1),\ T_2 \in sens(M_R)$? YES!

An Example of SPN Semantics Rule.

All transitions both firable and sensitized by the residual marking are fired.

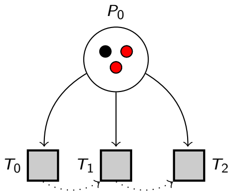
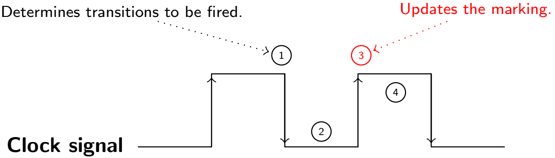


Figure: At state s.

$s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$

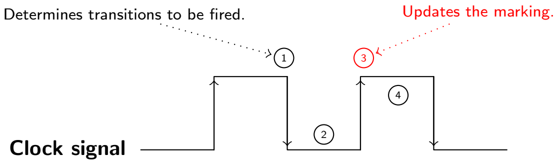
- $T_0,\ T_1 \in Fired'$
- $T_2 \in Fired'$?
- $M = (P_0, 3),\ T_2 \in firable(s)$? YES!
- $M_R = (P_0, 1),\ T_2 \in sens(M_R)$? YES!
- Then, according to rule 2 of SPN semantics: $T_2 \in Fired'$

SPN State Changing Relation (Rising Edge).



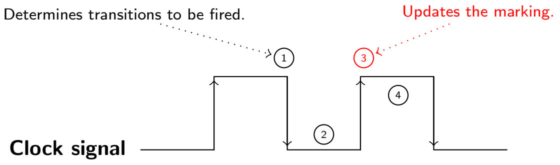
- $s = (Fired, M) \overset{\uparrow clock}{\rightsquigarrow} s' = (Fired, M')$:

SPN State Changing Relation (Rising Edge).



- $s = (Fired, M) \overset{\uparrow clock}{\rightsquigarrow} s' = (Fired, M')$:
  - M' is the new marking resulting from the firing of all transitions contained in Fired, i.e.:

SPN State Changing Relation (Rising Edge).



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

SPN Semantics in Coq.

```
Inductive SpnSemantics (spn : Spn) (s s' : SpnState) : Clock → Prop :=
| SpnSemantics_falling_edge :
    (* Rules 1, 2 and 3 *)
    ...  → SpnSemantics spn s s' falling_edge
| SpnSemantics_rising_edge :
    (* Ensures the consistency of spn, s and s'. *)
    IsWellDefinedSpn spn →
    IsWellDefinedSpnState spn s →
    IsWellDefinedSpnState spn s' →
    (* Fired stays the same between state s and s'. *)
    s.(fired) = s'.(fired) →
    (* Rule 4 of SPN semantics. *)
    (forall (p : Place) (n : nat),
    (p, n) ∈ s.(marking) →
    (p, n - (presum spn p s.(fired)) + (postsum spn p s.(fired))) ∈ s'.(marking)) →
    SpnSemantics spn s s' rising_edge.
```

- `s.(marking)` expresses the marking at state `s`.
- Markings are list of couples *(place, number of tokens)*.

SPN Token Player Program.

SPN Token Player Program.

- Implementation of the SPN semantics rules.

SPN Token Player Program.

- Implementation of the SPN semantics rules.
- Computes the evolution of a given SPN from initial state s_0 to state s_n, where n is the number of evolution cycles.

SPN Token Player Program.

- Implementation of the SPN semantics rules.
- Computes the evolution of a given SPN from initial state s_0 to state s_n, where n is the number of evolution cycles.
- Gives us confidence in our implementation of SPN semantics.

An Algorithm for one cycle of evolution.

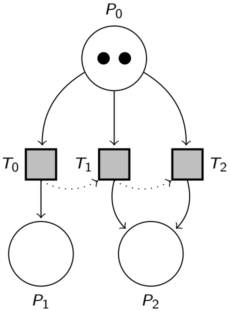
Data: spn, an SPN. s, the state of spn at the beginning of the clock cycle.
Result: A couple of SPN states, s' and s'', results of the evolution of spn from state s.

```
begin
    fired_transitions ← []
    /* Phase 1, falling edge of the clock.                                */
    foreach priority_group in spn.priority_groups do
        resid_m ← s.marking
        foreach trans in priority_group do
            if is_firable(trans, s) and is_sensitized(trans, resid_m) then
                update_residual_marking(trans, resid_m)
                push_back(trans, fired_transitions)
    s' ← make_state(fired_transitions, s.marking)
    /* Phase 2, rising edge of the clock.                                 */
    new_marking ← s'.marking
    foreach trans in fired_transitions do
        update_marking_pre(trans, new_marking)
        update_marking_post(trans, new_marking)
    s'' ← make_state(s'.fired, new_marking)
    return (s', s'')
```

Algorithm 1: cycle(spn, s)

Execution on An Example.

Falling edge phase.



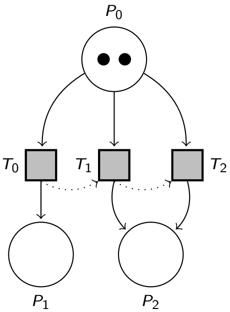

```
fired_transitions ← []

foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)

s' ← make_state(fired_transitions, s.marking)
```

$\texttt{s} = (\textit{fired}, \textit{marking})$ with $\texttt{s.marking} = (P_0, 2),\ (P_1, 0),\ (P_2, 0)$

$\texttt{priority\_groups} = [\ \ [T_0, T_1, T_2]\ \ ]$

Execution on An Example.

Falling edge phase.

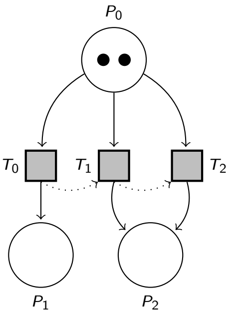



```
fired_transitions ← []

foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)

s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{priority\_groups} = [\ [T_0, T_1, T_2]\ ]$$
$$\texttt{fired\_transitions} = []$$

Execution on An Example.

Falling edge phase.

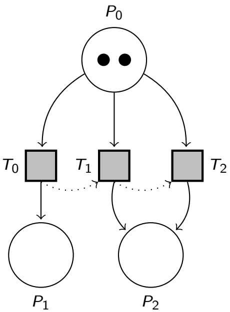



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)
s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{priority\_groups} = [\ \ [T_0, T_1, T_2]\ \ ]$$
$$\texttt{fired\_transitions} = []$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$

Execution on An Example.

Falling edge phase.

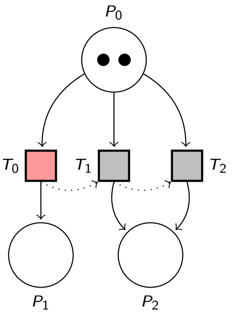



```
fired_transitions ← []

foreach priority_group in spn.priority_groups do
    resid_m ← s.marking

    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)

s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{fired\_transitions} = []$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 2),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.

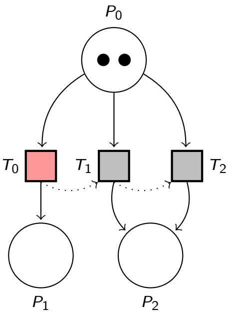



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)
s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{fired\_transitions} = []$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 2),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.



fired_transitions ← []

foreach *priority_group in* spn.priority_groups **do**

  resid_m ← s.marking

  foreach *trans in priority_group* **do**

    if `is_firable`(*trans*, s) **and** `is_sensitized`(*trans*, resid_m) **then**

      `update_residual_marking`(*trans*, resid_m)

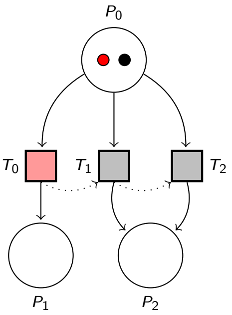
      `push_back`(*trans*, fired_transitions)

s' ← `make_state`(fired_transitions, s.marking)

$$\texttt{fired\_transitions} = []$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 2),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.



fired_transitions ← []

foreach *priority_group in* spn.priority_groups **do**

resid_m ← s.marking

foreach *trans in priority_group* **do**

if `is_firable`(*trans*, s) **and** `is_sensitized`(*trans*, resid_m) **then**

`update_residual_marking`(*trans*, resid_m)

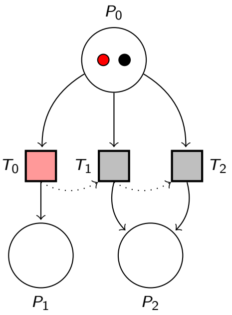
`push_back`(*trans*, fired_transitions)

s' ← `make_state`(fired_transitions, s.marking)

$$\texttt{fired\_transitions} = []$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 1),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.

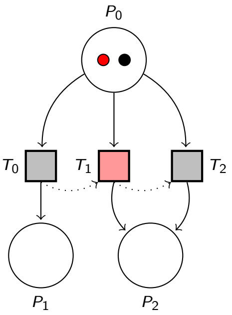



```
fired_transitions ← []

foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)

s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{fired\_transitions} = [T_0]$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 1),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.

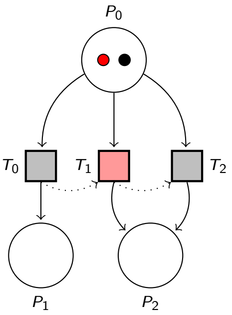



```
fired_transitions ← []

foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)

s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{fired\_transitions} = [T_0]$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 1),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.

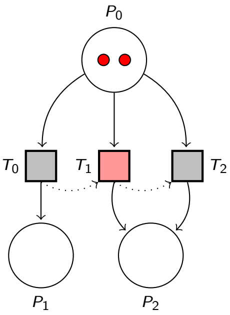



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)
s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{fired\_transitions} = [T_0]$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 1),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.

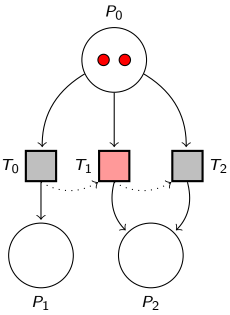



```
fired_transitions ← []

foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)

s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{fired\_transitions} = [T_0]$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 0),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.

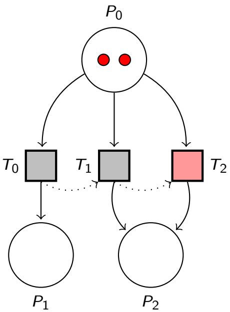



```
fired_transitions ← []

foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)

s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 0),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.

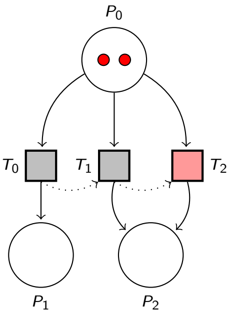



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)
s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 0),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.

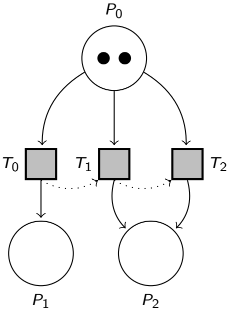



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)
s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{priority\_group} = [T_0, T_1, T_2]$$
$$\texttt{resid\_m} = (P_0, 0),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Falling edge phase.

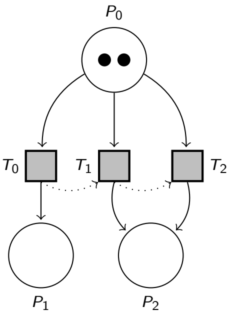



```
fired_transitions ← []

foreach priority_group in spn.priority_groups do
    resid_m ← s.marking

    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)

s' ← make_state(fired_transitions, s.marking)
```

$$\texttt{s'} = ([T_0, T_1], [(P_0, 2),\ (P_1, 0),\ (P_2, 0)])$$

Execution on An Example.

Rising edge phase.



new_marking ← s'.marking

foreach *trans in* fired_transitions **do**
    update_marking_pre(*trans*, new_marking)
    update_marking_post(*trans*, new_marking)

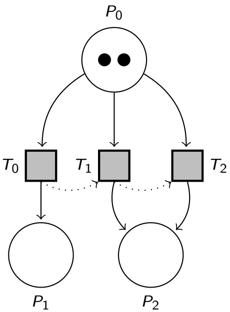
s" ← make_state(s'.fired, new_marking)

return *(*s', s"*)*

$$\texttt{s'} = ([T_0, T_1], [(P_0, 2),\ (P_1, 0),\ (P_2, 0)])$$
$$\texttt{fired\_transitions} = [T_0, T_1]$$

Execution on An Example.

Rising edge phase.



new_marking ← s'.marking

foreach *trans in* fired_transitions **do**

    update_marking_pre(*trans*, new_marking)
    update_marking_post(*trans*, new_marking)

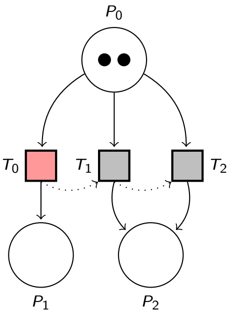
s" ← make_state(s'.fired, new_marking)

return *(*s', s"*)*

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{new\_marking} = (P_0, 2),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Rising edge phase.



new_marking ← s'.marking

foreach *trans in* fired_transitions **do**

    update_marking_pre(*trans*, new_marking)

    update_marking_post(*trans*, new_marking)

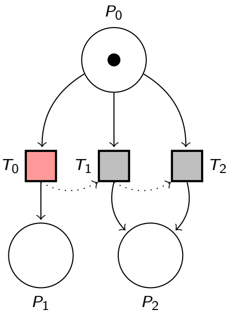
s" ← make_state(s'.fired, new_marking)

return *(*s', s"*)*

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{new\_marking} = (P_0, 2),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Rising edge phase.



new_marking ← s'.marking

foreach *trans in* fired_transitions **do**

    update_marking_pre(*trans*, new_marking)

    update_marking_post(*trans*, new_marking)

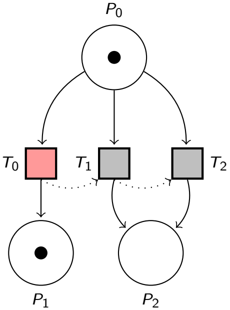
s" ← make_state(s'.fired, new_marking)

return *(*s', s"*)*

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{new\_marking} = (P_0, 1),\ (P_1, 0),\ (P_2, 0)$$

Execution on An Example.

Rising edge phase.



new_marking ← s'.marking

foreach *trans in* fired_transitions **do**

    update_marking_pre(*trans*, new_marking)
    update_marking_post(*trans*, new_marking)

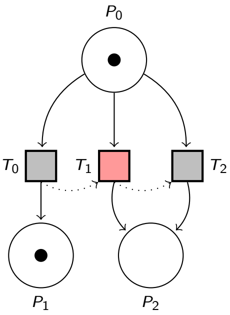
s" ← make_state(s'.fired, new_marking)

return *(*s', s"*)*

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{new\_marking} = (P_0, 1),\ (P_1, 1),\ (P_2, 0)$$

Execution on An Example.

Rising edge phase.



new_marking ← s'.marking

foreach *trans in* fired_transitions **do**

    update_marking_pre(*trans*, new_marking)
    update_marking_post(*trans*, new_marking)

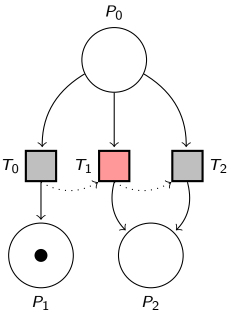
s" ← make_state(s'.fired, new_marking)

return *(*s', s"*)*

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{new\_marking} = (P_0, 1),\ (P_1, 1),\ (P_2, 0)$$

Execution on An Example.

Rising edge phase.

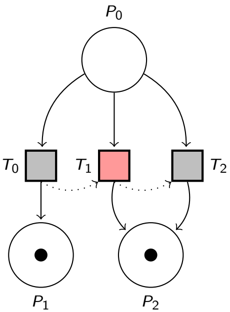



```
new_marking ← s'.marking

foreach trans in fired_transitions do
    update_marking_pre(trans, new_marking)
    update_marking_post(trans, new_marking)

s" ← make_state(s'.fired, new_marking)

return (s', s")
```

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{new\_marking} = (P_0, 0),\ (P_1, 1),\ (P_2, 0)$$

Execution on An Example.

Rising edge phase.



new_marking ← s'.marking

foreach *trans in* fired_transitions **do**

  update_marking_pre(*trans*, new_marking)

  update_marking_post(*trans*, new_marking)

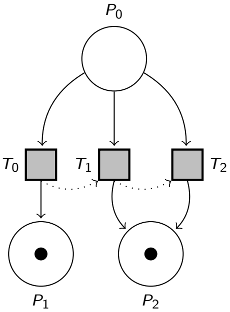
s" ← make_state(s'.fired, new_marking)

return *(*s', s"*)*

$$\texttt{fired\_transitions} = [T_0, T_1]$$
$$\texttt{new\_marking} = (P_0, 0),\ (P_1, 1),\ (P_2, 1)$$

Execution on An Example.

Rising edge phase.



new_marking ← s'.marking

foreach *trans in* fired_transitions **do**

```
update_marking_pre(trans, new_marking)
update_marking_post(trans, new_marking)
```

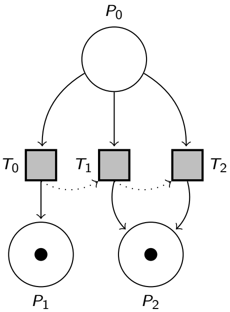
s'' ← `make_state`(s'.fired, new_marking)

return *(*s', s''*)*

$$\texttt{s''} = ([T_0, T_1], [(P_0, 0),\ (P_1, 1),\ (P_2, 1)])$$

Execution on An Example.

Rising edge phase.



new_marking ← s'.marking

foreach *trans in* fired_transitions **do**

    update_marking_pre(*trans*, new_marking)

    update_marking_post(*trans*, new_marking)

s" ← make_state(s'.fired, new_marking)

return *(*s', s"*)*

$$\texttt{s'} = ([T_0, T_1], [(P_0, 2),\ (P_1, 0),\ (P_2, 0)])$$
$$\texttt{s''} = ([T_0, T_1], [(P_0, 0),\ (P_1, 1),\ (P_2, 1)])$$

Correctness/Completeness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
$\texttt{cycle spn s = (s', s'')} \Rightarrow s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s''$.

Correctness/Completeness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
`cycle spn s = (s', s'')` $\Rightarrow$ `s` $\overset{\downarrow clock}{\rightsquigarrow}$ `s'` $\overset{\uparrow clock}{\rightsquigarrow}$ `s''`.

Theorem (Completeness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
`s` $\overset{\downarrow clock}{\rightsquigarrow}$ `s'` $\overset{\uparrow clock}{\rightsquigarrow}$ `s''` $\Rightarrow$ `cycle spn s = (s', s'')`.

Conclusion.

Conclusion.

Context.

- Formal verification of a model-to-text transformation from HILECOP PNs to VHDL.
- First step: model the semantics of HILECOP PNs (SITPNs).

Conclusion.

Context.

- Formal verification of a model-to-text transformation from HILECOP PNs to VHDL.
- First step: model the semantics of HILECOP PNs (SITPNs).

Done.

Model the semantics of SPNs (subclass of HILECOP PNs).

Conclusion.

Context.

- Formal verification of a model-to-text transformation from HILECOP PNs to VHDL.
- First step: model the semantics of HILECOP PNs (SITPNs).

Done.

Model the semantics of SPNs (subclass of HILECOP PNs).

On Going.

Add time, interpretation and macroplaces to SPNs semantics.

Thank you for your attention!

Bibliography.

D. Andreu, D. Guiraud, and S. Guillaume.
A distributed architecture for activating the peripheral nervous system.
Journal of Neural Engineering, 6(2):18, Feb. 2009.

H. Leroux.
Méthodologie de conception d'architectures numériques complexes: du formalisme à l'implémentation en passant par l'analyse, préservation de la conformité. Application aux neuroprothèses. (A methodology for the design of complex digital architectures: from formalism to implementation, taking into account formal analysis. Application to neuroprosthetics).
PhD thesis, Montpellier 2 University, France, 2014.

X. Leroy.
Formal Verification of a Realistic Compiler.
Communications of the ACM (CACM), 52(7):107–115, July 2009.

I. Merzoug.
Validation formelle des systèmes numériques critiques : génération de l'espace d'états de réseaux de Petri exécutés en synchrone. (Formal validation of critical digital systems : generation of state space of Petri nets executed in synchronous).
PhD thesis, University of Montpellier, France, 2018.

The Coq Development Team.
Coq, version 8.9.0.
Inria, Jan. 2019.
http://coq.inria.fr/.

Coq Implementation of the SPN Token Player.

```
Definition spn_cycle (spn : Spn) (starting_state : SpnState) :
    option (SpnState * SpnState) :=
    (* Computes the transitions to be fired. *)
    match spn_falling_edge spn starting_state with
    | Some inter_state ⇒
      (* Updates the marking. *)
      match spn_rising_edge spn inter_state with
      | Some final_state ⇒ Some (inter_state, final_state)
      | None ⇒ None
      end
    | None ⇒ None
    end.
```

Figure: The SPN Token Player Program in Coq.

Coq Implementation of the SPN Token Player.

```
Definition spn_cycle (spn : Spn) (starting_state : SpnState) :
    option (SpnState * SpnState) :=
    (* Computes the transitions to be fired. *)
    match spn_falling_edge spn starting_state with
    | Some inter_state ⇒
      (* Updates the marking. *)
      match spn_rising_edge spn inter_state with
      | Some final_state ⇒ Some (inter_state, final_state)
      | None ⇒ None
      end
    | None ⇒ None
    end.
```

Figure: The SPN Token Player Program in Coq.

- `match` checks the result of function calls.

Coq Implementation of the SPN Token Player.

```
Definition spn_cycle (spn : Spn) (starting_state : SpnState) :
    option (SpnState * SpnState) :=
    (* Computes the transitions to be fired. *)
    match spn_falling_edge spn starting_state with
    | Some inter_state ⇒
      (* Updates the marking. *)
      match spn_rising_edge spn inter_state with
      | Some final_state ⇒ Some (inter_state, final_state)
      | None ⇒ None
      end
    | None ⇒ None
    end.
```

Figure: The SPN Token Player Program in Coq.

- `match` checks the result of function calls.
- Functions return `Some value` or `None` (error case).

Reminder on Correctness and Completeness.

- Let X, Y be two types.
- Let $P \in X \rightarrow Y$ be a program, that takes $x \in X$ as an input value and returns some $y \in Y$.
- Let $S \in X \rightarrow Y \rightarrow \{\top, \perp\}$ be the specification of program P. S is a predicate that takes x and y as input values and return True or False.

Definition (Correctness)

A program P is said to be correct regarding its specification if $\forall x \in X, y \in Y,\ P(x) = y \Rightarrow S(x, y)$

Definition (Completeness)

A program P is said to be complete regarding its specification if $\forall x \in X, y \in Y,\ S(x, y) \Rightarrow P(x) = y$

Correctness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
spn_cycle spn s = Some (s', s'') $\Rightarrow s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s''$.

Correctness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
spn_cycle spn s = Some (s', s'') $\Rightarrow$ *s* $\overset{\downarrow clock}{\rightsquigarrow}$ *s'* $\overset{\uparrow clock}{\rightsquigarrow}$ *s''.*

Lemma (Falling Edge Correct)

$\forall$ *(spn : Spn) (s s' : SpnState), which are well-defined,*
spn_falling_edge spn s = Some s' $\Rightarrow$ *s* $\overset{\downarrow clock}{\rightsquigarrow}$ *s'.*

Correctness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
$\texttt{spn\_cycle spn s = Some (s', s'')} \Rightarrow s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s''$.

Lemma (Falling Edge Correct)

$\forall$ *(spn : Spn) (s s' : SpnState), which are well-defined,*
$\texttt{spn\_falling\_edge spn s = Some s'} \Rightarrow s \overset{\downarrow clock}{\rightsquigarrow} s'$.

Falling Edge Correct Proof.

- Induction on the priority groups of `spn`.
- With the help of other lemmas:
  1. `is_sensitized(t, M)` $\Leftrightarrow t \in sens(M)$
  2. `is_firable(t, s)` $\Leftrightarrow t \in firable(s)$
  3. `spn_falling_edge` computes a proper residual marking.
  4. ...

□

Correctness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
$\texttt{spn\_cycle spn s = Some (s', s'')} \Rightarrow s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s''$.

Lemma (Rising Edge Correct)

$\forall$ *(spn : Spn) (s s' : SpnState), which are well-defined,*
$\texttt{spn\_rising\_edge spn s = Some s'} \Rightarrow s \overset{\uparrow clock}{\rightsquigarrow} s'$.

Correctness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
`spn_cycle spn s = Some (s', s'')` $\Rightarrow s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s''$.

Lemma (Rising Edge Correct)

$\forall$ *(spn : Spn) (s s' : SpnState), which are well-defined,*
`spn_rising_edge spn s = Some s'` $\Rightarrow s \overset{\uparrow clock}{\rightsquigarrow} s'$.

Rising Edge Correct Proof.

- Induction on the list of transitions to be fired of state `s`.
- With the help of other lemmas:
  1. `update_marking_pre(t, M) = Some M'`
     $\Leftrightarrow M' = M - \sum_{t_i \in Fired} pre(t_i)$
  2. `update_marking_post(t, M) = Some M'`
     $\Leftrightarrow M' = M + \sum_{t_i \in Fired} post(t_i)$
  3. ...

□

Completeness of The SPN Token Player.

Theorem (Completeness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
$s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s'' \Rightarrow$ *spn_cycle spn s = Some (s', s'').*

Completeness of The SPN Token Player.

Theorem (Completeness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
$s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s'' \Rightarrow$ *spn_cycle spn s = Some (s', s'').*

Lemma (Falling Edge Complete)

$\forall$ *(spn : Spn) (s s' : SpnState), which are well-defined,*
$s \overset{\downarrow clock}{\rightsquigarrow} s' \Rightarrow$ *spn_falling_edge spn s = Some s'.*

Completeness of The SPN Token Player.

Theorem (Completeness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*
$s \overset{\downarrow clock}{\rightsquigarrow} s', \overset{\uparrow clock}{\rightsquigarrow} s'' \Rightarrow$ *spn_cycle spn s = Some (s', s'').*

Lemma (Falling Edge Complete)

$\forall$ *(spn : Spn) (s s' : SpnState), which are well-defined,*
$s \overset{\downarrow clock}{\rightsquigarrow} s' \Rightarrow$ *spn_falling_edge spn s = Some s'.*

Lemma (Rising Edge Complete)

$\forall$ *(spn : Spn) (s s' : SpnState), which are well-defined,*
$s \overset{\uparrow clock}{\rightsquigarrow} s' \Rightarrow$ *spn_rising_edge spn s = Some s'.*