Formal Verification of HILECOP.

A Process to Design and Implement Critical Digital Systems.

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

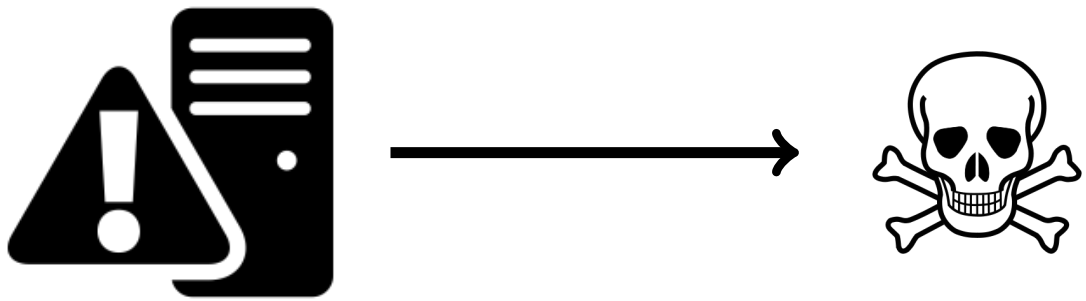
[2]NEURINNOV, Montpellier, France
David.Andreu@neurinnov.com

June 21, 2019

Context.

CRITICAL DIGITAL SYSTEMS (CDS)?

CRITICAL DIGITAL SYSTEMS (CDS)



CRITICAL DIGITAL SYSTEMS (CDS)

Avionics

Medicine

Automotive

Critical Digital Systems: Design and Implementation.

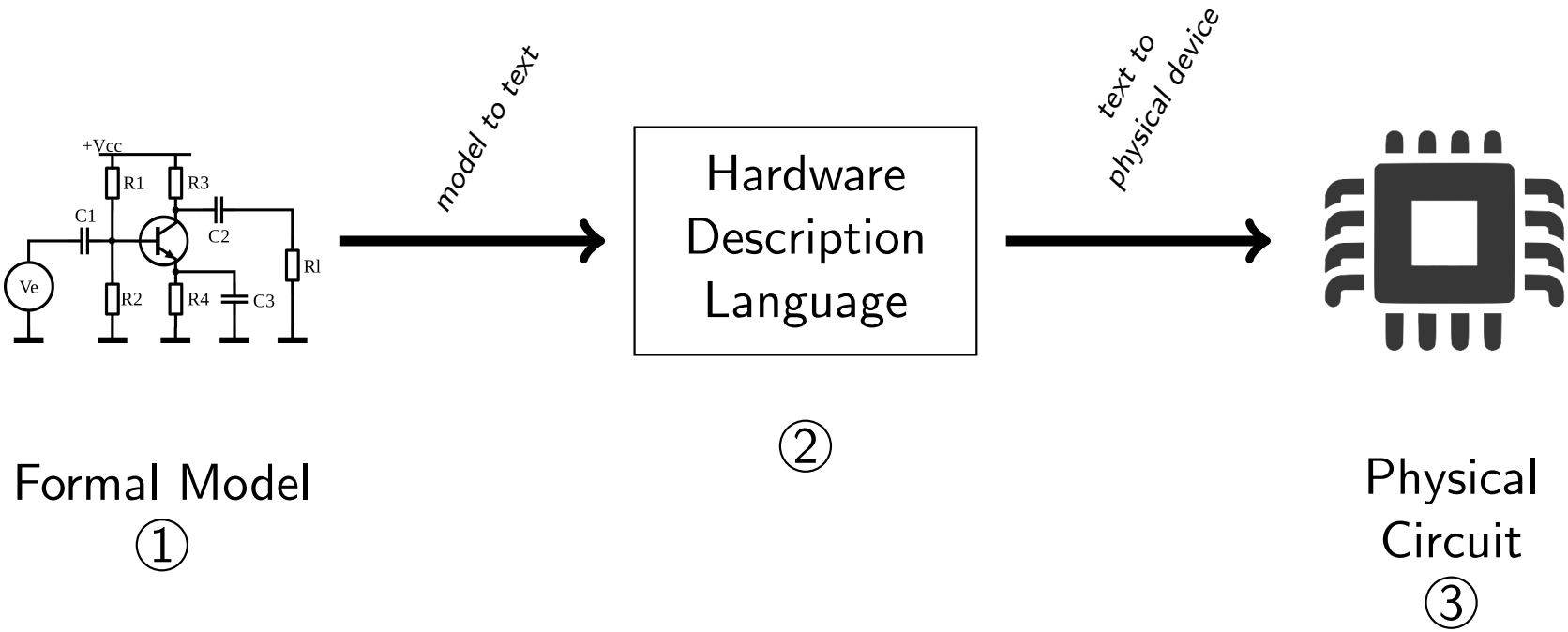


Figure: A Meta-process to Design and Implement CDS.

HILECOP: A Process to Design and Implement CDS.

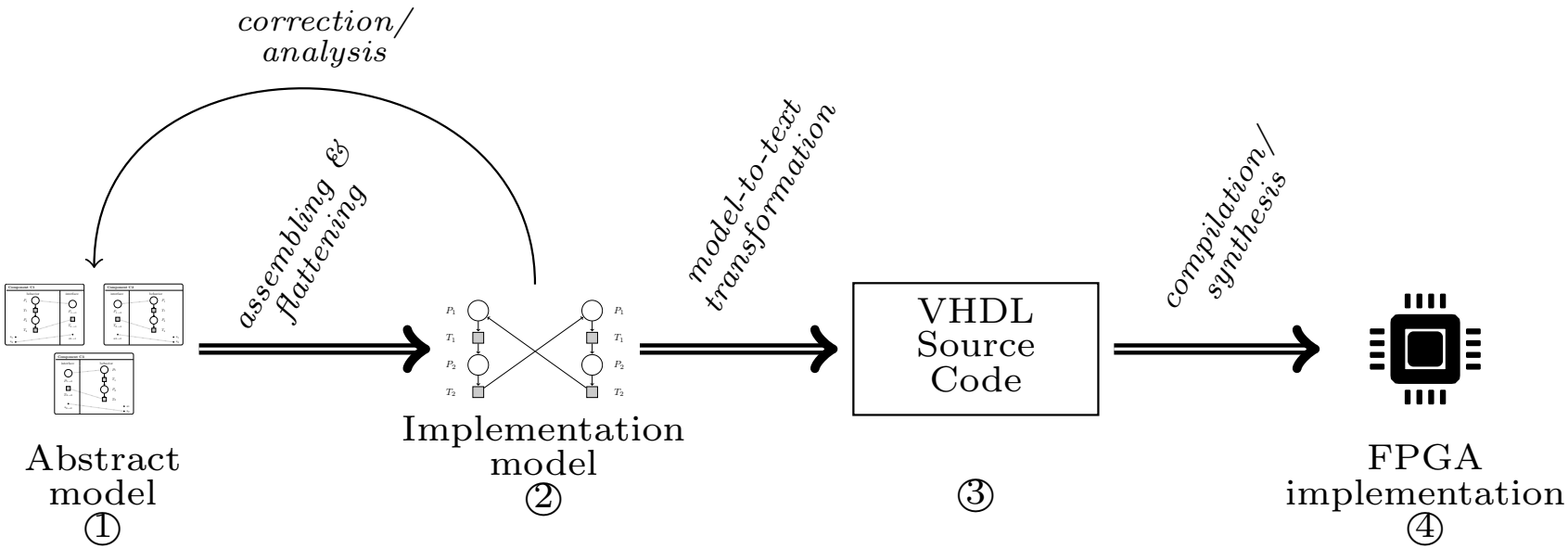


Figure: Workflow of the HILECOP Methodology, developed at INRIA (CAMIN Team) [1].

Formal Methods for HILECOP.

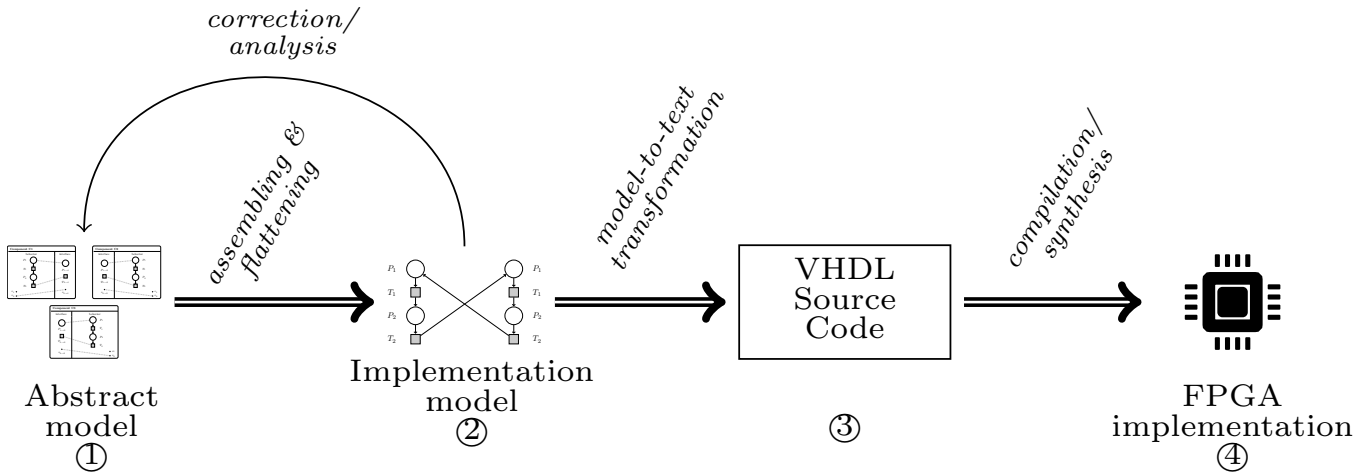


Figure: Workflow of the HILECOP Methodology, developed at Inria (CAMIN Team) [1].

Verification of HILECOP.

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

Formal Methods for HILECOP.

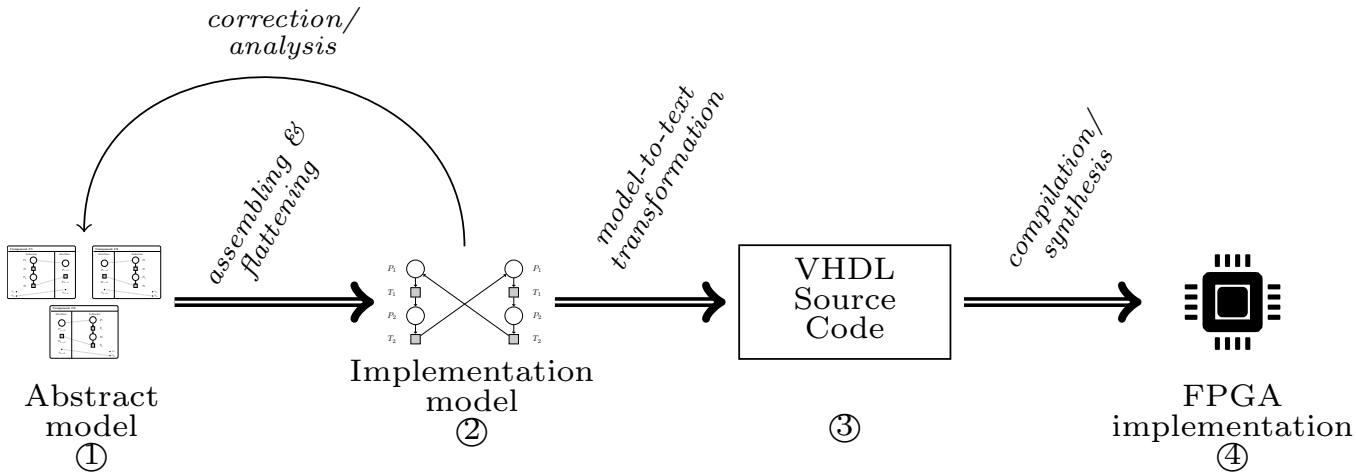


Figure: Workflow of the HILECOP Methodology, developed at Inria (CAMIN Team) [1].

Verification of HILECOP.

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

Purpose of Our Work.

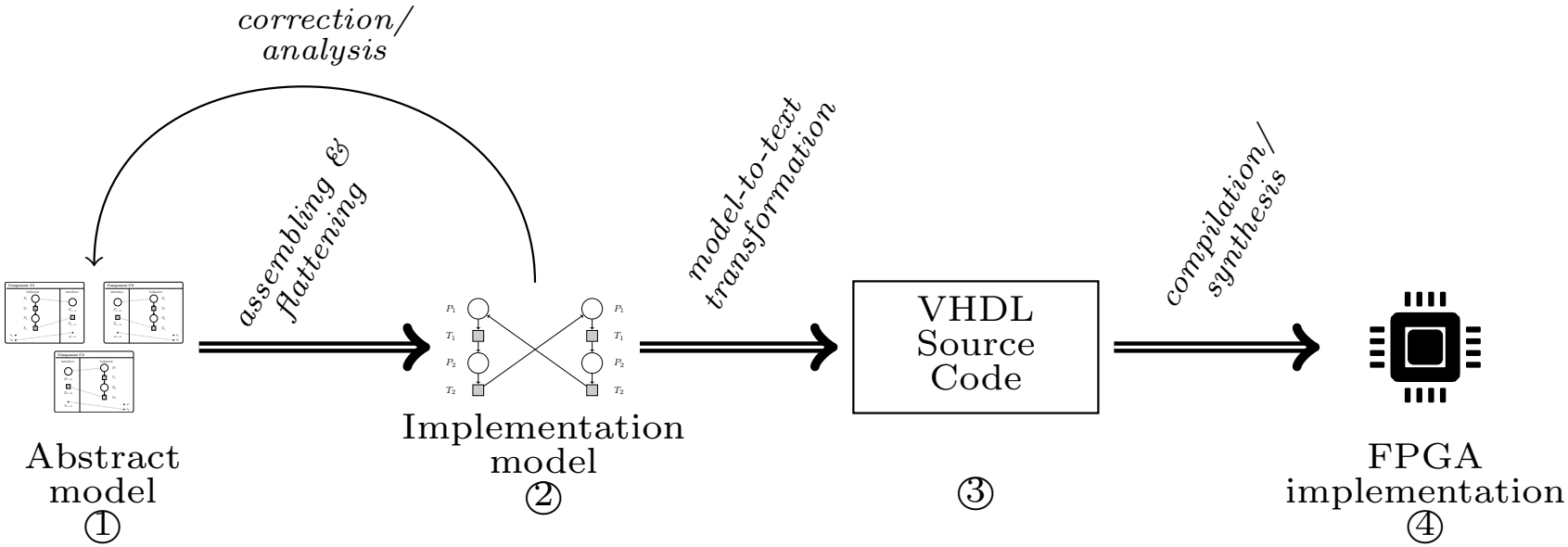


Figure: Workflow of the HILECOP Methodology.

Purpose of Our Work.

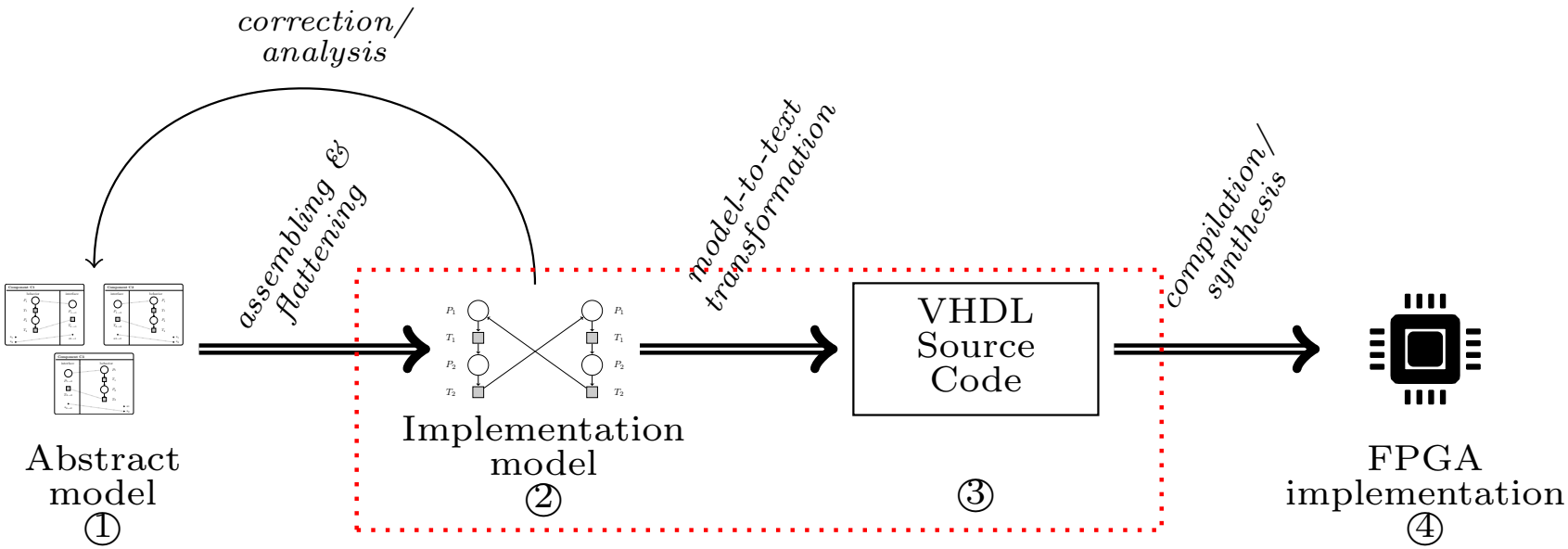
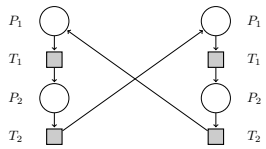


Figure: Workflow of the HILECOP Methodology.

Purpose of Our Work.



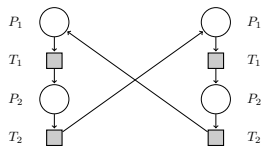
Implementation model

model-to-text transformation

VHDL Source Code

Figure: Part of the HILECOP workflow subject to verification.

Purpose of Our Work.



Implementation model

model-to-text transformation

VHDL Source Code

Figure: Part of the HILECOP workflow subject to verification.

Goal. Proof of behavior preservation.

Purpose of Our Work.

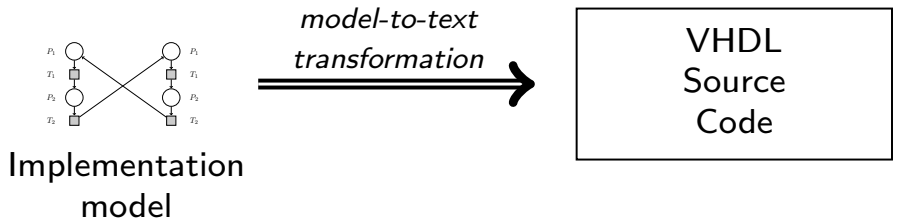


Figure: Part of the HILECOP workflow subject to verification.

Proof steps.

Inspired by *CompCert*, a formally verified C compiler [2], written with the Coq proof assistant [3]:

Purpose of Our Work.

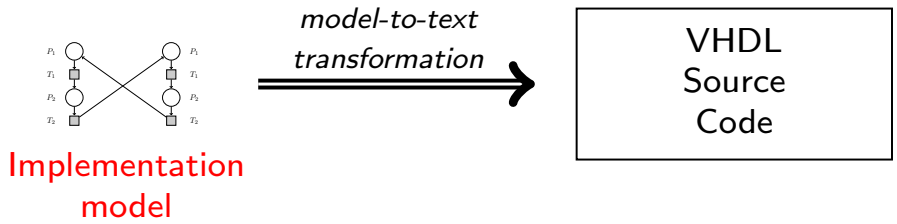


Figure: Part of the HILECOP workflow subject to verification.

Proof steps.

Inspired by *CompCert*, a formally verified C compiler [2], written with the Coq proof assistant [3]:

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

Purpose of Our Work.

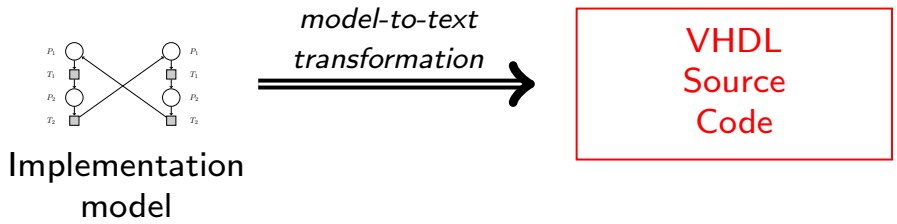


Figure: Part of the HILECOP workflow subject to verification.

Proof steps.

Inspired by *CompCert*, a formally verified C compiler [2], written with the Coq proof assistant [3]:

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

Purpose of Our Work.

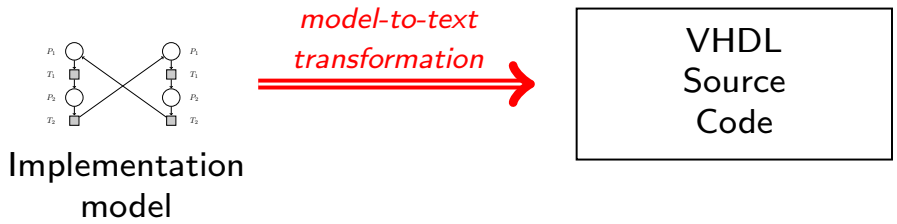


Figure: Part of the HILECOP workflow subject to verification.

Proof steps.

Inspired by *CompCert*, a formally verified C compiler [2], written with the Coq proof assistant [3]:

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 and prove behavior preservation.

A Reminder on the Coq proof assistant.

About.

- Developed by INRIA and CNAM teams since 1984.
- 1984: first contribution by Thierry Coquand and Gérard Huet.
- 1991: Christine Paulin extends the language with the *Calculus of Inductive Constructions*.

Coq: A bird with two legs.

- Generic Functional Programming Language.
- Proof language.

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).

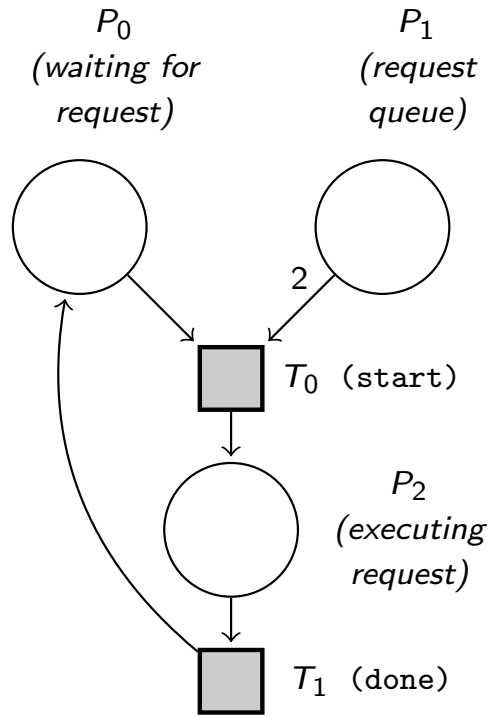
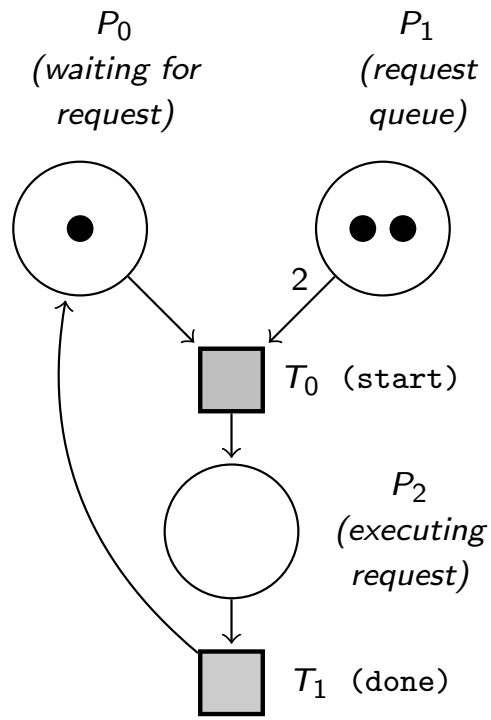


Figure: A request execution system modeled with a Petri net.

Marking and Transition Firing.

- Marking: current state of the system.
- Sensitization: a transition t is ready to be fired.

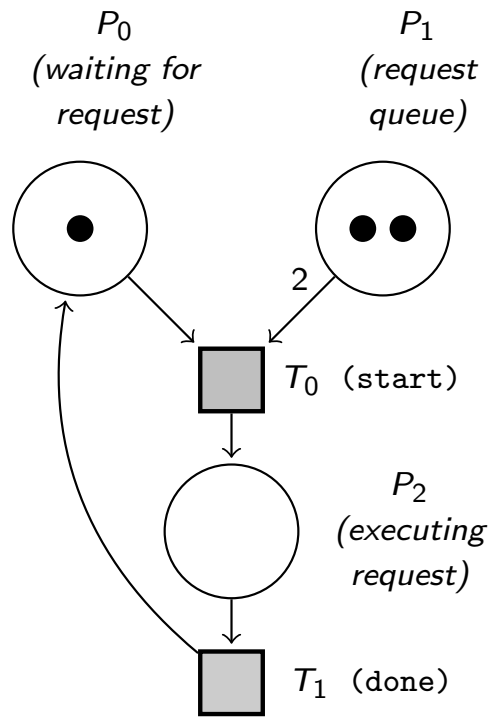


Marking and Transition Firing.

- Marking: current state of the system.
- Sensitization: a transition t is ready to be fired.

Transition firing.

- $M = (P_0, 1),\ (P_1, 2),\ (P_2, 0)$

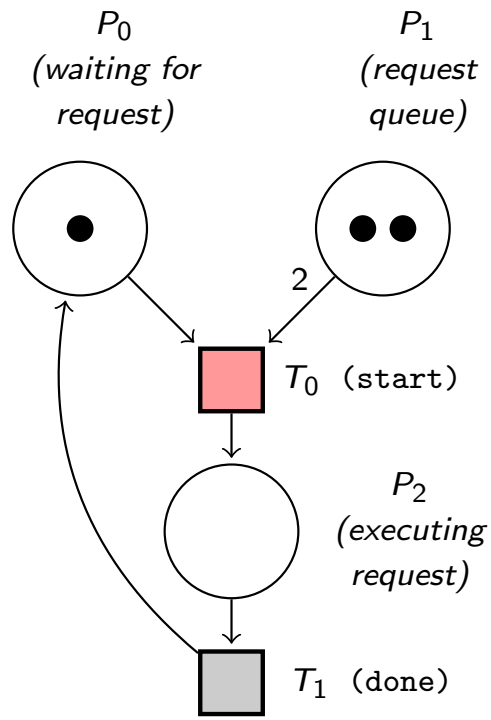


Marking and Transition Firing.

- Marking: current state of the system.
- Sensitization: a transition t is ready to be fired.

Transition firing.

- $M = (P_0, 1),\ (P_1, 2),\ (P_2, 0)$
- T_0 is sensitized $\Rightarrow$ T_0 is fired.

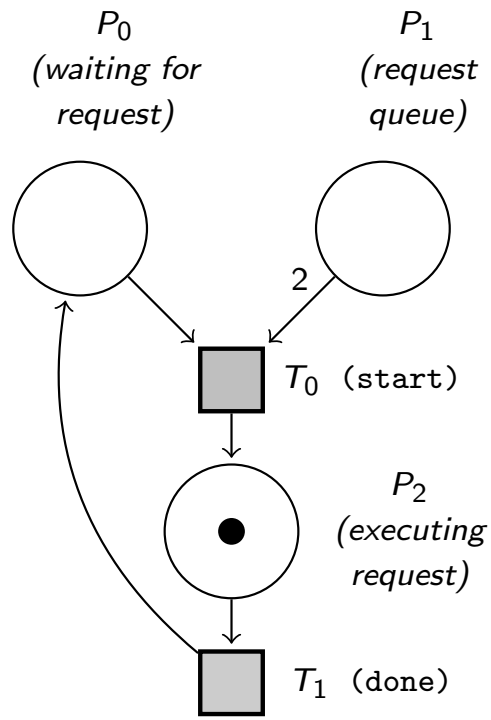


Marking and Transition Firing.

- Marking: current state of the system.
- Sensitization: a transition t is ready to be fired.

Transition firing.

- $M = (P_0, 1),\ (P_1, 2),\ (P_2, 0)$
- T_0 is sensitized $\Rightarrow$ T_0 is fired.
- $M' = (P_0, 0),\ (P_1, 0),\ (P_2, 1)$



HILECOP High-Level Models.

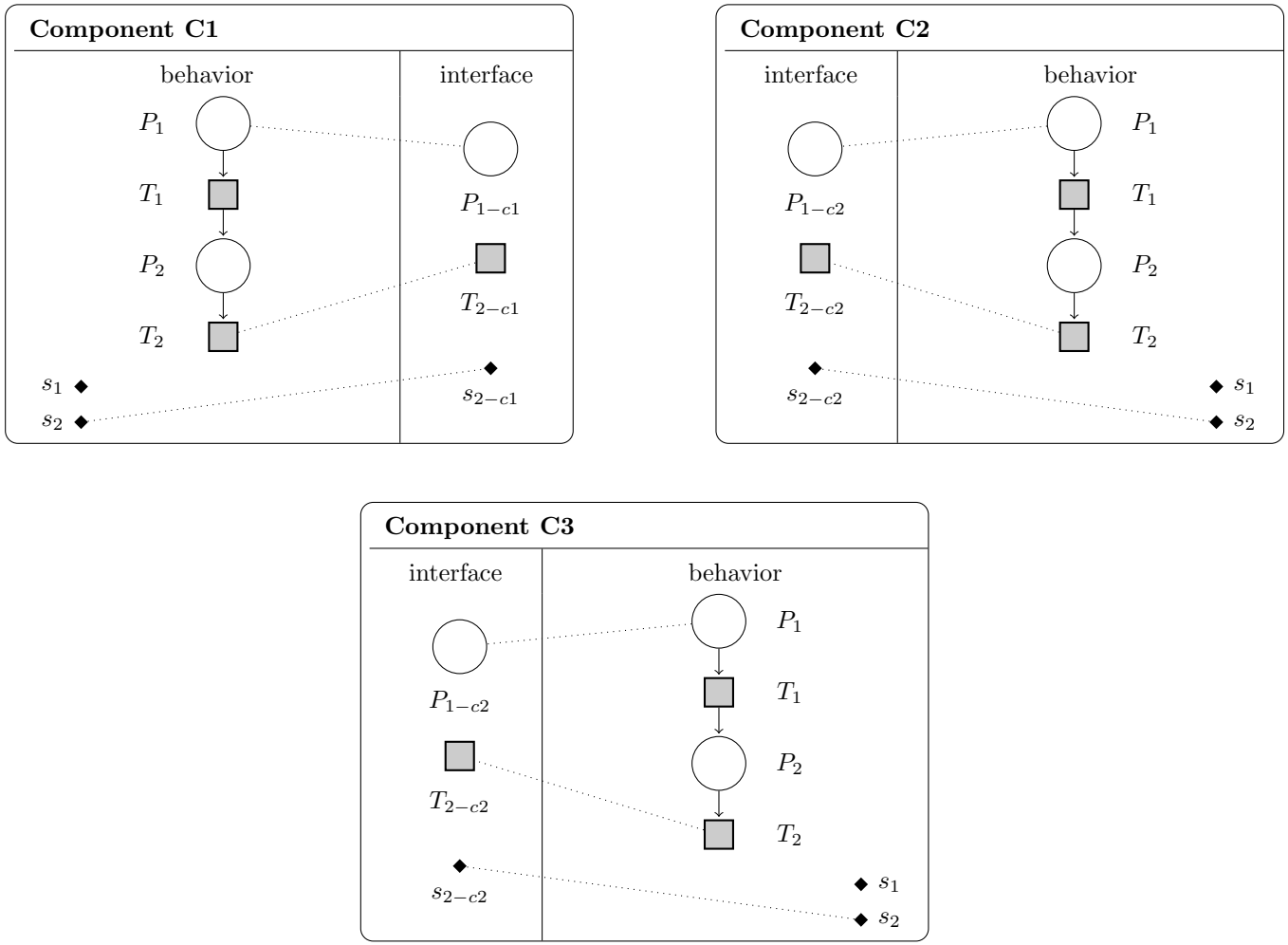


Figure: An Example of HILECOP high-level model at the first stage of the workflow.

HILECOP High-Level Models.

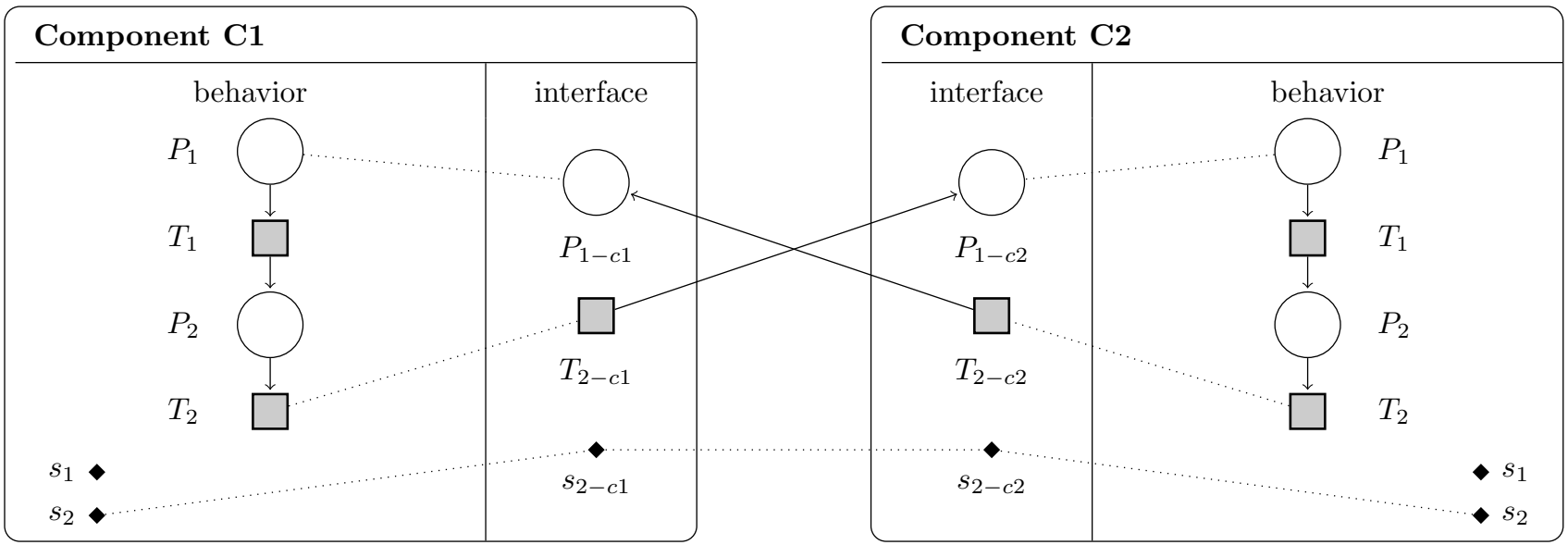


Figure: Component assembling in a HILECOP high-level model.

HILECOP High-Level Models.

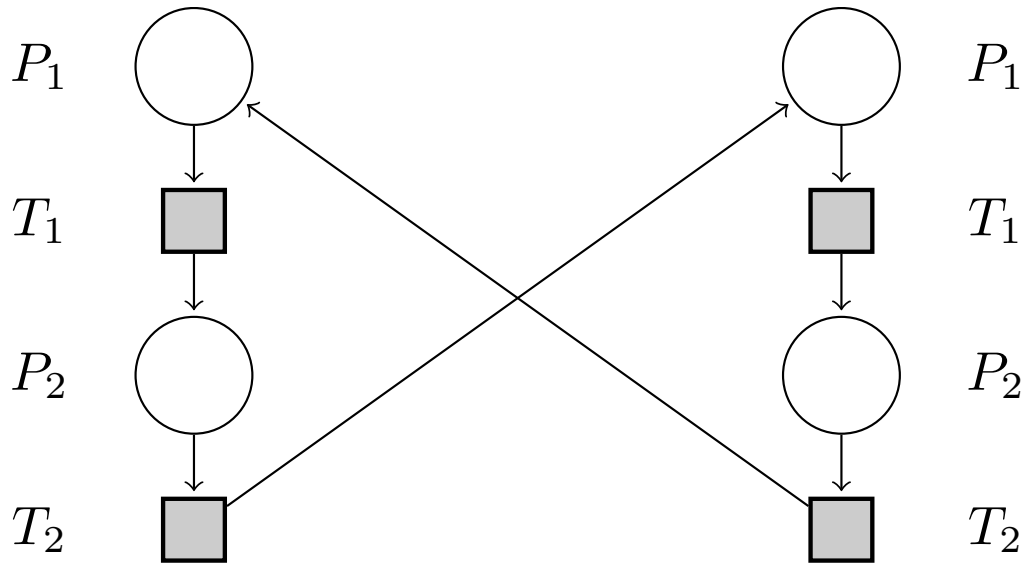


Figure: Flattened version of the model.

Remark.

The flattened model corresponds to the *implementation-ready* model, input of the model-to-text transformation.

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

We will only present Synchronously executed (with priorities), generalized, extended Petri Nets (SPNs).

Generalized and extended PNs.

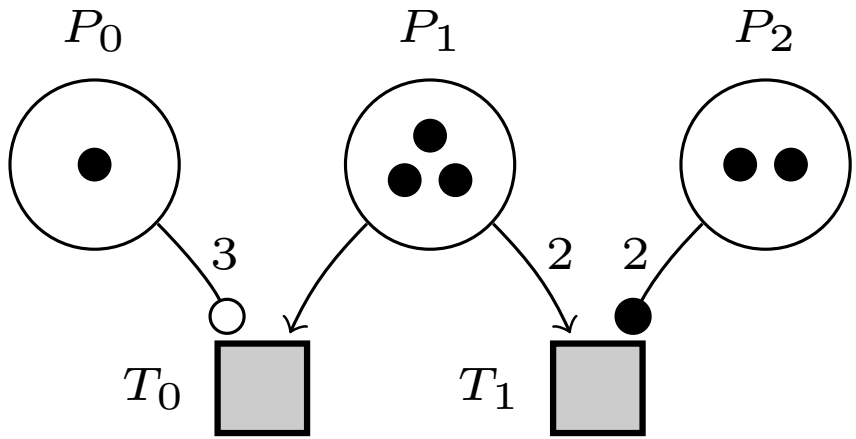
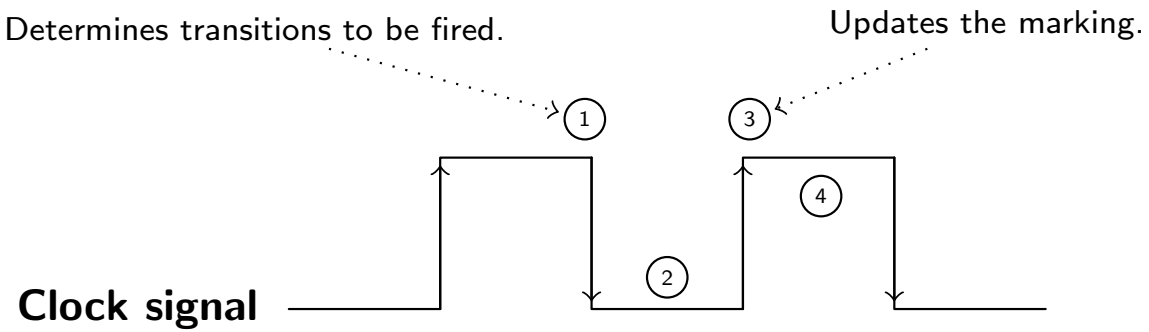
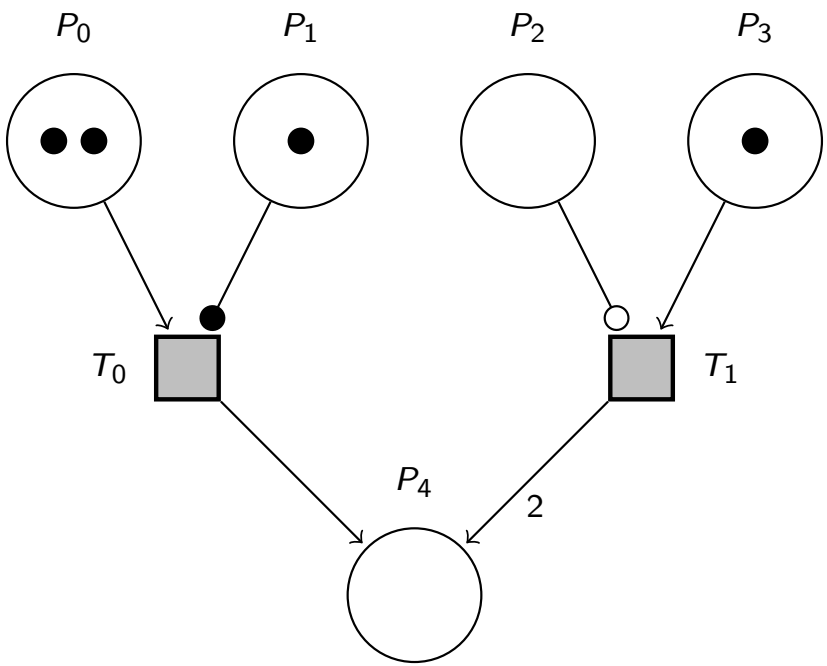


Figure: An example of extended, generalized PN.

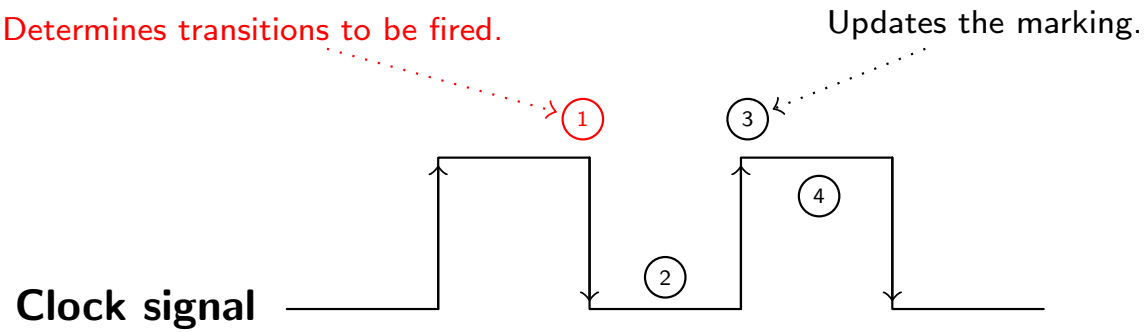
- Generalized: Edge weights $\in \mathbb{N}$.
- Extended: Inhibitor and test edges.

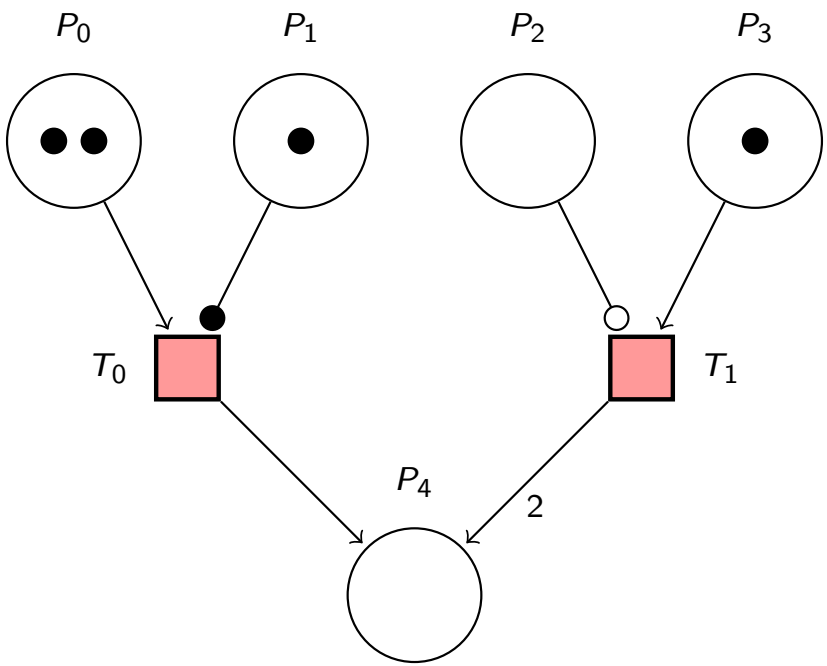
Synchronously Executed PNs.



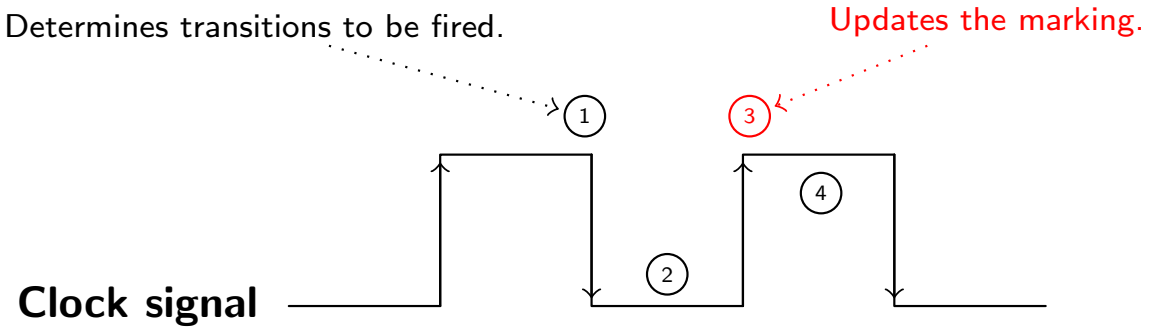


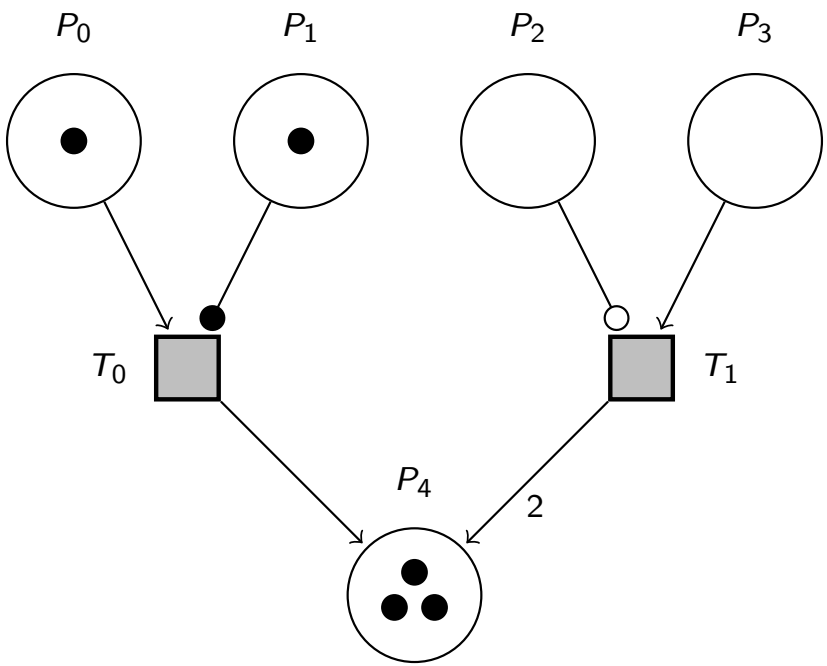
Synchronously Executed PNs.





Synchronously Executed PNs.





Conflicts and priorities.

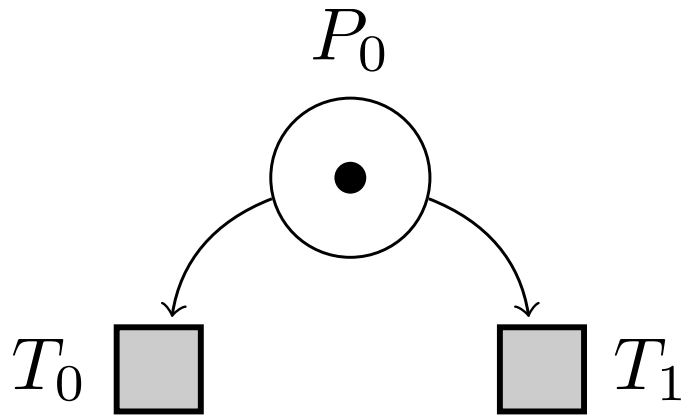


Figure: An Example of Conflict (Structural and Effective).

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.

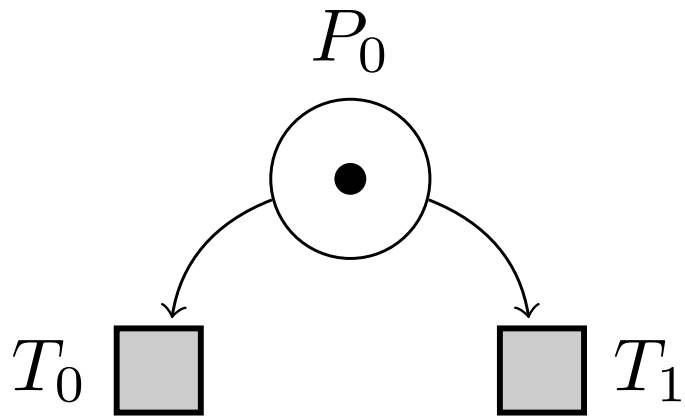


Figure: An Example of Conflict (Structural and Effective).

Which transition will be fired?

Conflicts and priorities.

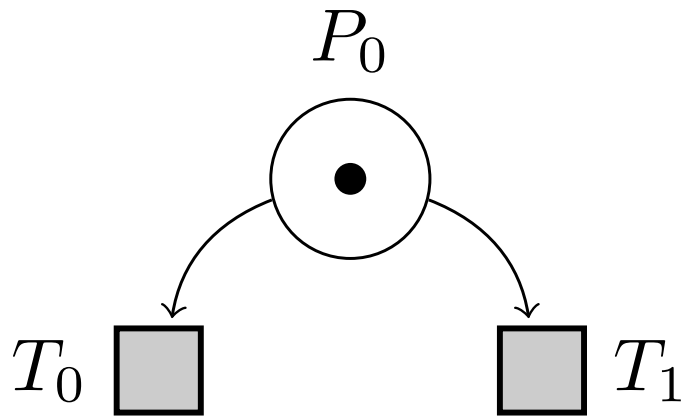


Figure: An Example of Conflict (Structural and Effective).

Which transition will be fired?

- If asynchronous execution: T_0 or T_1

Conflicts and priorities.

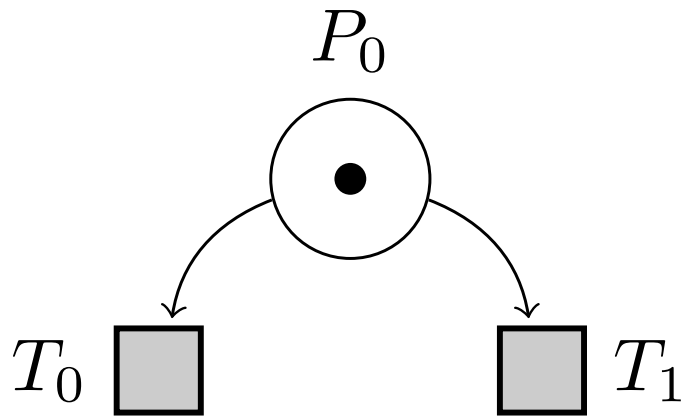


Figure: An Example of Conflict (Structural and Effective).

Which transition will be fired?

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

Conflicts and priorities.

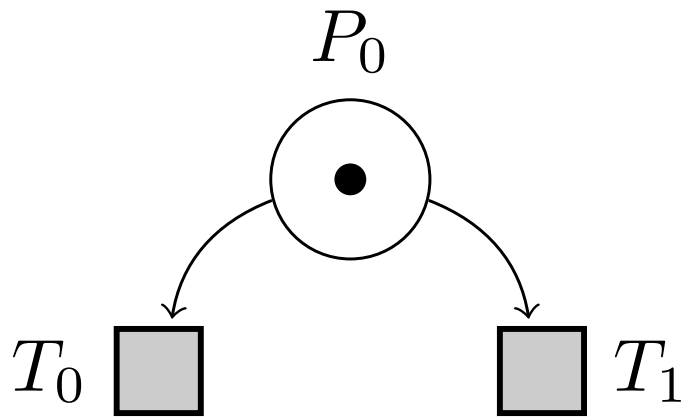


Figure: An Example of Conflict (Structural and Effective).

Which transition will be fired?

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

Conflicts and priorities.

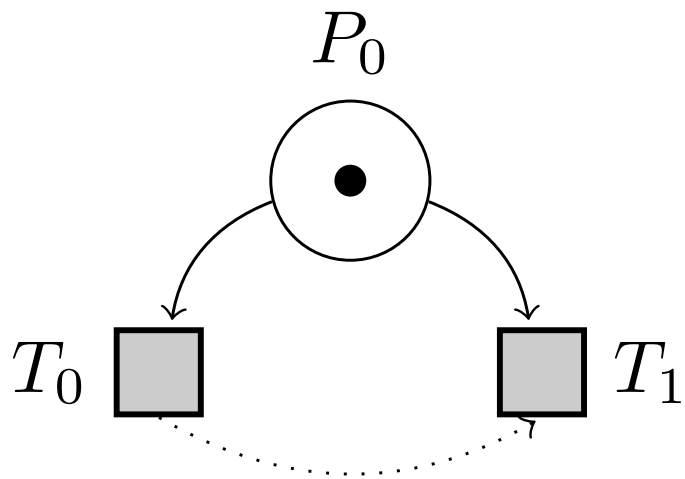


Figure: Resolving conflicts with priorities.

Priority relation.

T_0 has a higher firing priority than T_1.

Conflicts and priorities.

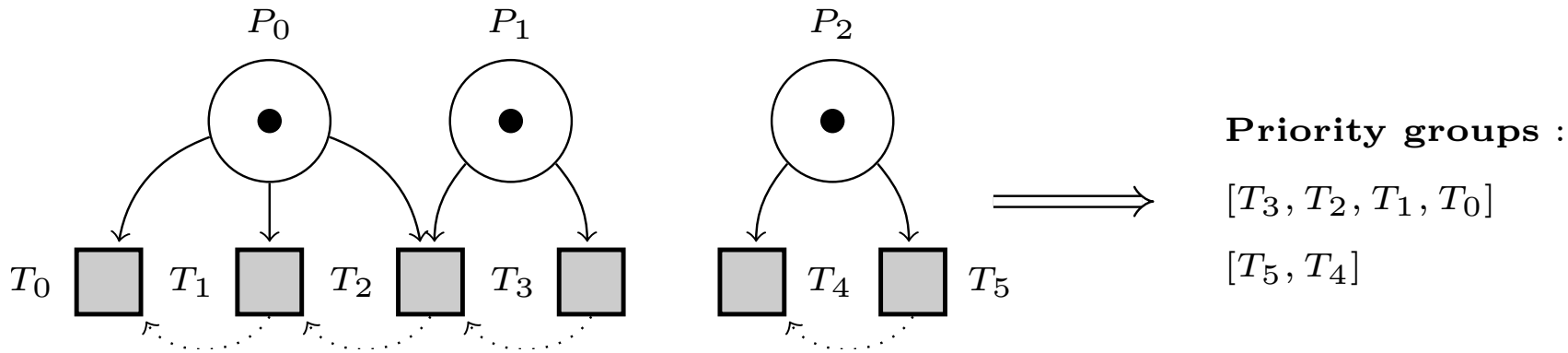


Figure: Determining priority groups in a PN.

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. $pre \in P \to T \to \mathbb{N}$.
4. $test \in P \to T \to \mathbb{N}$.
5. $inhib \in P \to T \to \mathbb{N}$.
6. $post \in T \to P \to \mathbb{N}$.
7. $M_0 \in P \to \mathbb{N}$, the initial marking of the SPN.
8. $clock \in \{\downarrow clock, \uparrow clock\}$.
9. $\succ$, the priority relation, which represents the firing priority between transitions of the same priority group.

Implementation of SPNs in Coq.

```
1  Structure Spn : Set :=
2    mk_Spn {
3       places : list Place;
4       transs : list Trans;
5       pre : Place → Trans → nat;
6       test : Place → Trans → nat;
7       inhib : Place → Trans → nat;
8       post : Trans → Place → nat;
9       initial_marking : Place → nat;
10      priority_groups : list (list Trans);
11      lneighbors : Trans → Neighbors;
12   }.
```

Figure: The Coq structure for SPNs.

Definitions and Notations.

Remark.

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

Definitions and Notations.

Remark.

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

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 $<P, T, pre, test, inhib, post, M_0, clock, \succ>$.

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)

- A transition $t \in sens(M)$, if $M \geq pre(t)$, and $M \geq test(t)$, and $M < inhib(t)$ or $inhib(t) = 0$.
- 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.

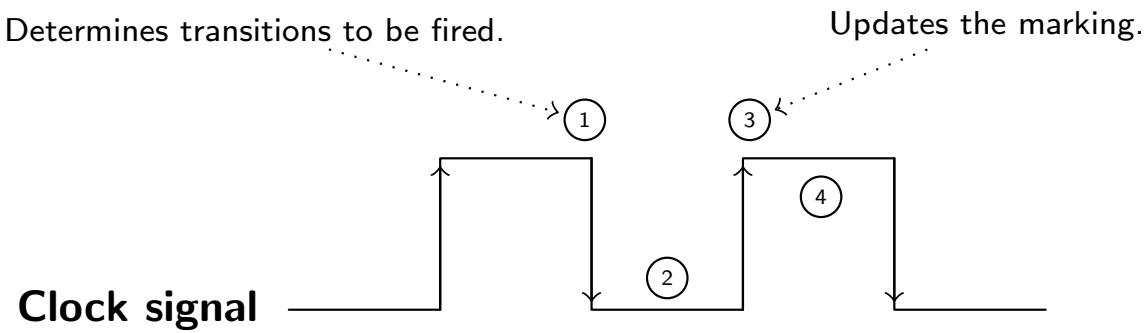
SPN Semantics.

Definition (SPN Semantics)

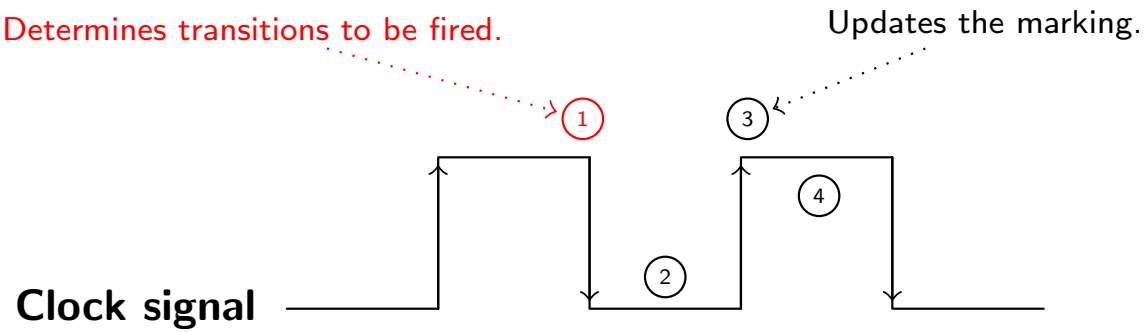
The semantics of an SPN is represented by the triplet $< S, s_0, \rightsquigarrow >$ where:

- 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 $s, s' \in S$ and $clk \in Clk$.

SPN State Changing Relation (Falling Edge).

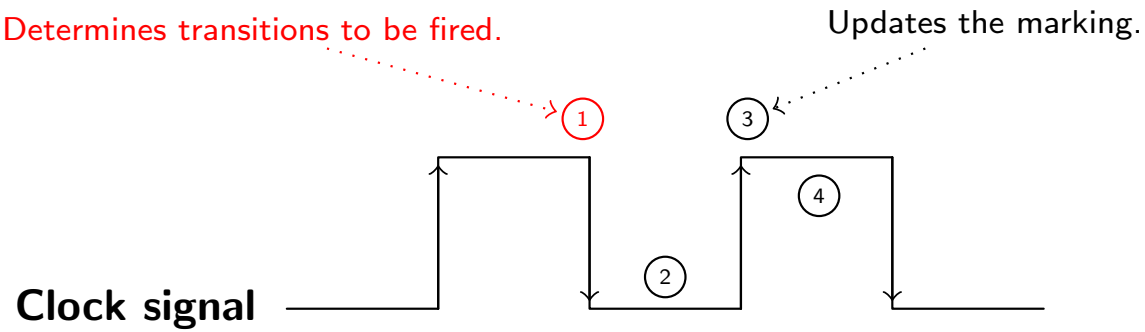


SPN State Changing Relation (Falling Edge).



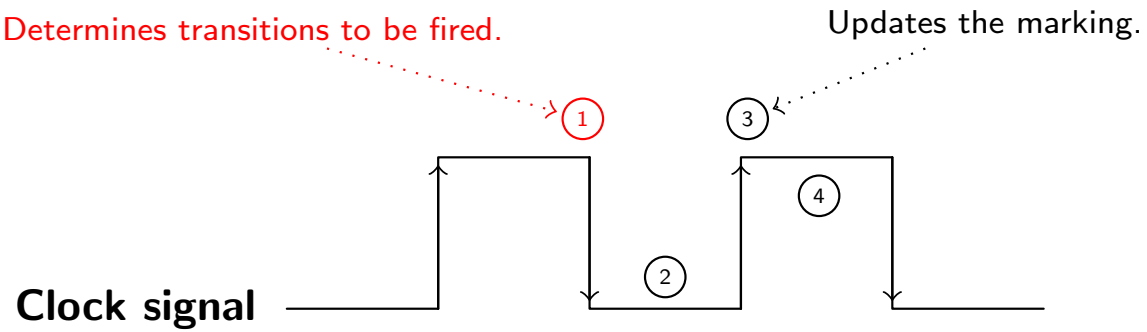
- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if $\downarrow clock = 1$ and:

SPN State Changing Relation (Falling Edge).



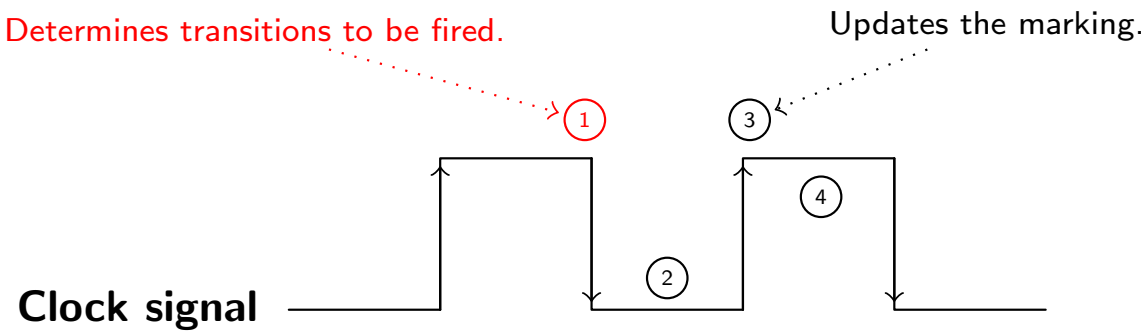
- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if $\downarrow clock = 1$ and:
  1. All transitions that are not firable are not fired, i.e.: $\forall t \in T,\ t \notin firable(s) \Rightarrow t \notin Fired'$.

SPN State Changing Relation (Falling Edge).



- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if $\downarrow clock = 1$ and:
  1. All transitions that are not firable are not fired, i.e.: $\forall t \in T,\ t \notin firable(s) \Rightarrow t \notin Fired'$.
  2. All transitions both firable and 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 $Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in Fired'\}$.

SPN State Changing Relation (Falling Edge).



- $s = (Fired, M) \overset{\downarrow clock}{\rightsquigarrow} s' = (Fired', M)$ if $\downarrow clock = 1$ and:
  1. All transitions that are not firable are not fired, i.e.: $\forall t \in T,\ t \notin firable(s) \Rightarrow t \notin Fired'$.
  2. All transitions both firable and 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 $Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in Fired'\}$.
  3. All firable transitions that are not sensitized by the residual marking are not fired, i.e.: $\forall t \in firable(s),\ t \notin sens\big(M - \sum_{t_i \in Pr(t)} pre(t_i)\big) \Rightarrow t \notin Fired'$.

An Example of SPN Semantics Rule.

$$\forall t \in \mathit{firable}(s),\ t \in \mathit{sens}\big(M - \textstyle\sum_{t_i \in Pr(t)} \mathit{pre}(t_i)\big) \Rightarrow t \in \mathit{Fired}',$$
$$\text{where } Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in \mathit{Fired}'\}$$

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

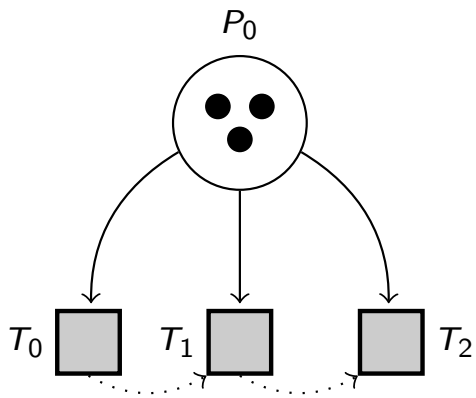


Figure: At state s.

An Example of SPN Semantics Rule.

$$\forall t \in \mathit{firable}(s),\ t \in \mathit{sens}\big(M - \textstyle\sum_{t_i \in Pr(t)} \mathit{pre}(t_i)\big) \Rightarrow t \in \mathit{Fired}',$$
$$\text{where } Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in \mathit{Fired}'\}$$

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

- $T_0,\ T_1 \in \mathit{Fired}'$

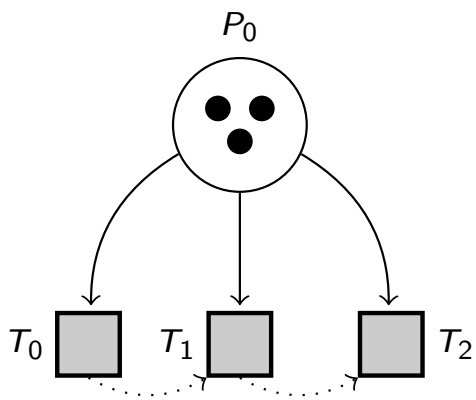


Figure: At state s.

An Example of SPN Semantics Rule.

$$\forall t \in \mathit{firable}(s),\ t \in \mathit{sens}\big(M - \textstyle\sum_{t_i \in Pr(t)} \mathit{pre}(t_i)\big) \Rightarrow t \in \mathit{Fired}',$$
$$\text{where } Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in \mathit{Fired}'\}$$

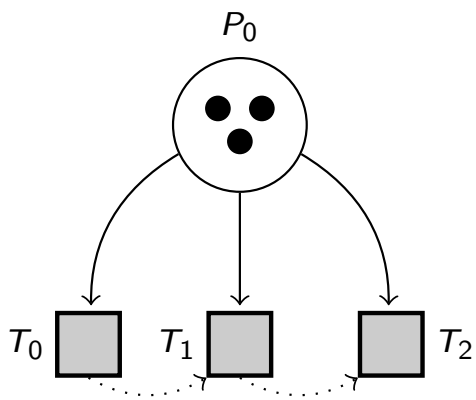


Figure: At state s.

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

- $T_0,\ T_1 \in \mathit{Fired}'$
- $T_2 \in \mathit{Fired}'$?

An Example of SPN Semantics Rule.

$$\forall t \in \mathit{firable}(s),\ t \in \mathit{sens}\big(M - \textstyle\sum_{t_i \in Pr(t)} \mathit{pre}(t_i)\big) \Rightarrow t \in \mathit{Fired}',$$
$$\text{where } Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in \mathit{Fired}'\}$$

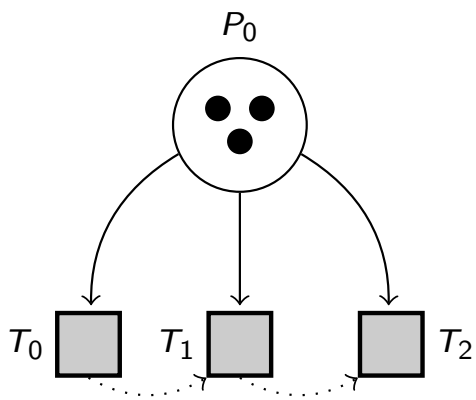


Figure: At state s.

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

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

An Example of SPN Semantics Rule.

$$\forall t \in \mathit{firable}(s),\ t \in \mathit{sens}\big(M - \textstyle\sum_{t_i \in Pr(t)} \mathit{pre}(t_i)\big) \Rightarrow t \in \mathit{Fired}',$$
$$\text{where } Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in \mathit{Fired}'\}$$

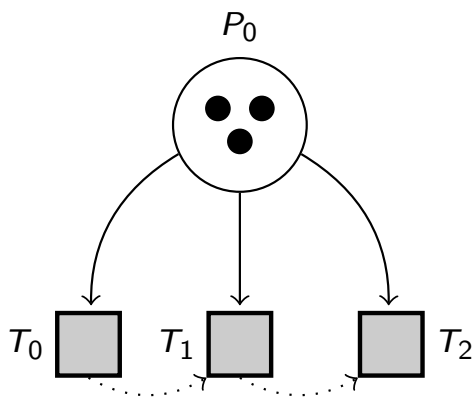


Figure: At state s.

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

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

An Example of SPN Semantics Rule.

$$\forall t \in \mathit{firable}(s),\ t \in \mathit{sens}\big(M - \textstyle\sum_{t_i \in Pr(t)} \mathit{pre}(t_i)\big) \Rightarrow t \in \mathit{Fired}',$$
$$\text{where } Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in \mathit{Fired}'\}$$

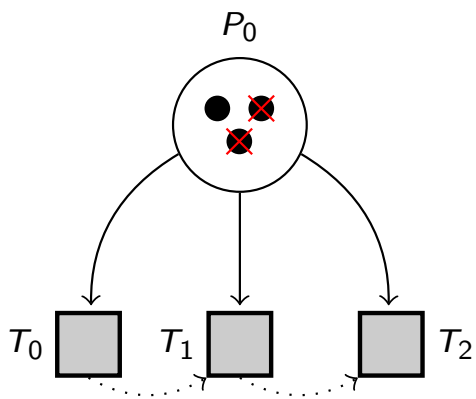


Figure: At state s.

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

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

An Example of SPN Semantics Rule.

$$\forall t \in \mathit{firable}(s),\ t \in \mathit{sens}\big(M - \textstyle\sum_{t_i \in Pr(t)} \mathit{pre}(t_i)\big) \Rightarrow t \in \mathit{Fired}',$$
$$\text{where } Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in \mathit{Fired}'\}$$

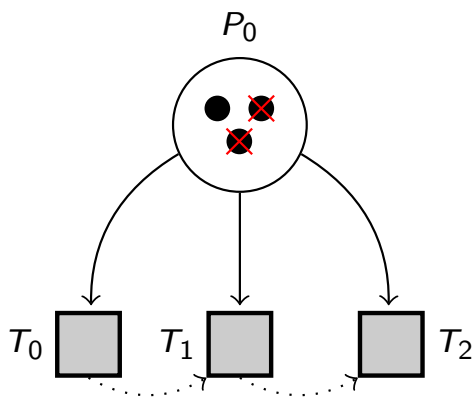


Figure: At state s.

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

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

An Example of SPN Semantics Rule.

$$\forall t \in \mathit{firable}(s),\ t \in \mathit{sens}\big(M - \textstyle\sum_{t_i \in Pr(t)} \mathit{pre}(t_i)\big) \Rightarrow t \in \mathit{Fired}',$$
$$\text{where } Pr(t) = \{t_i \mid t_i \succ t \wedge t_i \in \mathit{Fired}'\}$$

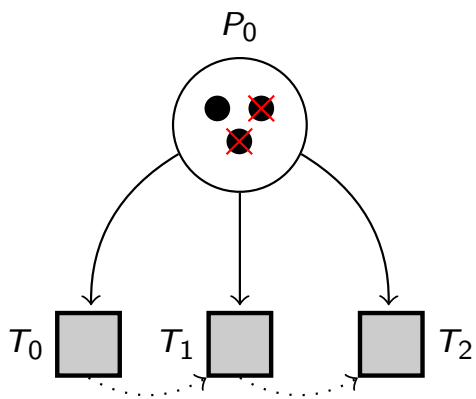
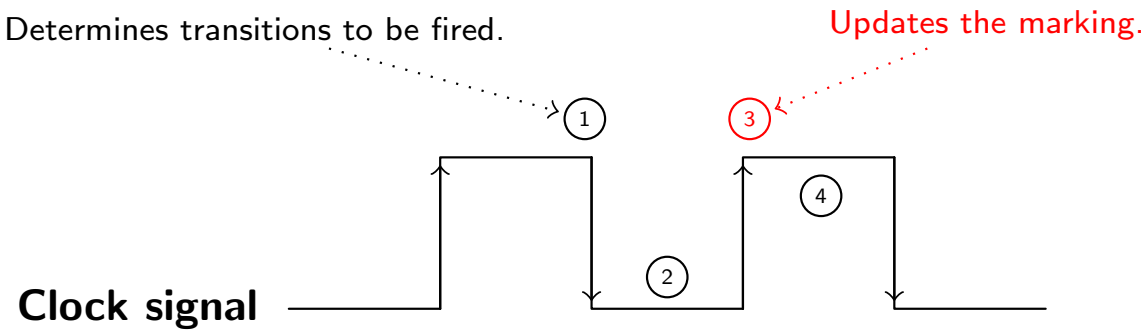


Figure: At state s.

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

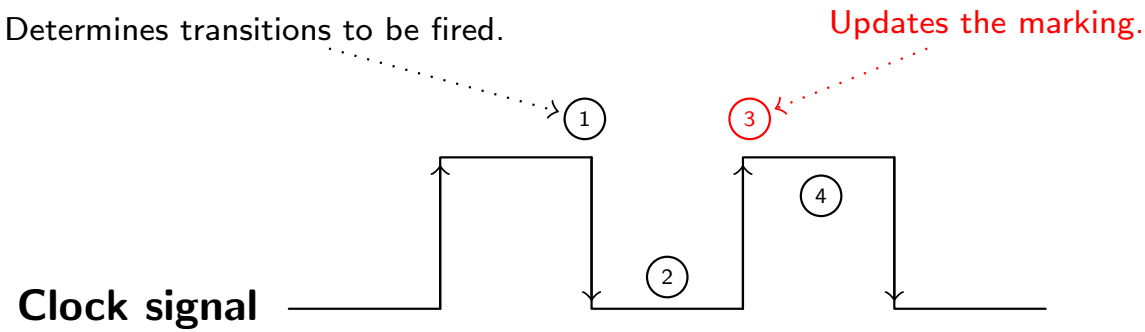
- $T_0,\ T_1 \in \mathit{Fired}'$
- $T_2 \in \mathit{Fired}'$?
- $M = (P_0, 3),\ T_2 \in \mathit{firable}(s)$? YES!
- $M_R = (P_0, 1),\ T_2 \in \mathit{sens}(M_R)$? YES!
- Then, according to rule 2 of SPN semantics: $T_2 \in \mathit{Fired}'$

SPN State Changing Relation (Rising Edge).



- $s = (Fired, M) \overset{\uparrow clock}{\rightsquigarrow} s' = (Fired, M')$ if $\uparrow clock = 1$ and:

SPN State Changing Relation (Rising Edge).



- $s = (Fired, M) \overset{\uparrow clock}{\rightsquigarrow} s' = (Fired, M')$ if $\uparrow clock = 1$ and:
  4. 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.

```
1  Inductive SpnSemantics (spn : Spn) (s s' : SpnState) : Clock → Prop :=
2  | SpnSemantics_falling_edge :
3      (* Rules 1, 2 and 3 *)
4      ...  → SpnSemantics spn s s' falling_edge
5  | SpnSemantics_rising_edge :
6      (* Ensures the consistency of spn, s and s'. *)
7      IsWellDefinedSpn spn →
8      IsWellDefinedSpnState spn s →
9      IsWellDefinedSpnState spn s' →
10     (* Fired stays the same between state s and s'. *)
11     s.(fired) = s'.(fired) →
12     (* Rule 4 of SPN semantics. *)
13     (forall (p : Place) (n : nat),
14         In (p, n) s.(marking) →
15         In (p, n − (pre_sum spn p s.(fired)) + (post_sum spn p s.(fired)))
16            s'.(marking)) → SpnSemantics spn s s' rising_edge.
```

Figure: The Semantics of SPNs in Coq.

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.
- Unformal way to verify the SPN semantics.

An Algorithm for one cycle of evolution.

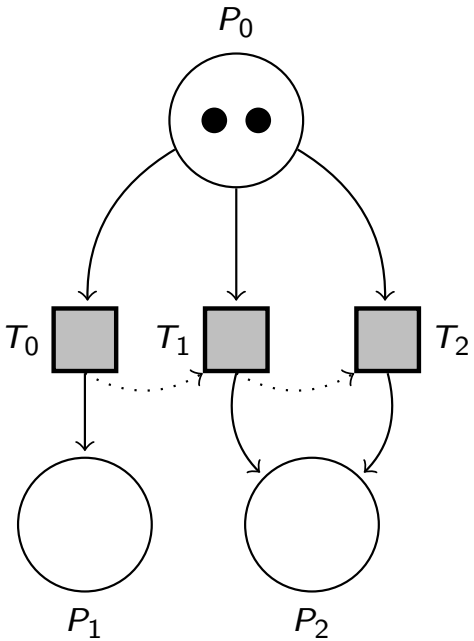
Data: spn, an SPN. s, the state of spn at the beginning of the clock cycle.
Result: A couple of SPN states, s' and s'', results of the evolution of spn from state s.

```
1  begin
2      fired_transitions ← []
       /* Phase 1, falling edge of the clock.                                  */
3      foreach priority_group in spn.priority_groups do
4          resid_m ← s.marking
5          foreach trans in priority_group do
6              if is_firable(trans, s) and is_sensitized(trans, resid_m) then
7                  update_residual_marking(trans, resid_m)
8                  push_back(trans, fired_transitions)
9      s' ← make_state(fired_transitions, s.marking)
       /* Phase 2, rising edge of the clock.                                   */
10     new_marking ← s'.marking
11     foreach trans in fired_transitions do
12         update_marking_pre(trans, new_marking)
13         update_marking_post(trans, new_marking)
14     s'' ← make_state(s'.fired, new_marking)
15     return (s', s'')
```

Algorithm 1: cycle(spn, s)

Execution on An Example.

Falling edge phase.



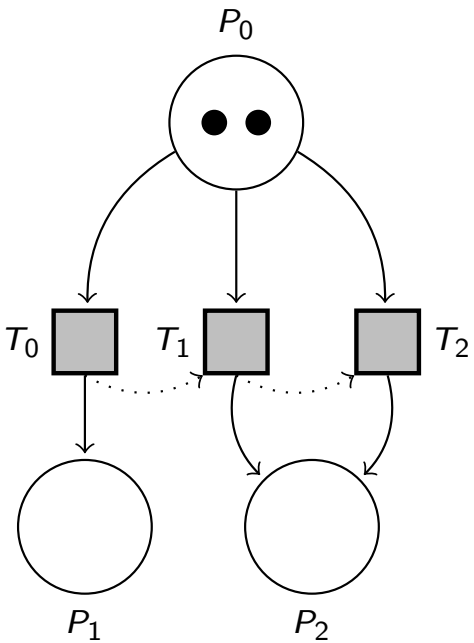

```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

$\texttt{s} = (\mathit{fired}, \mathit{marking})$ 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.

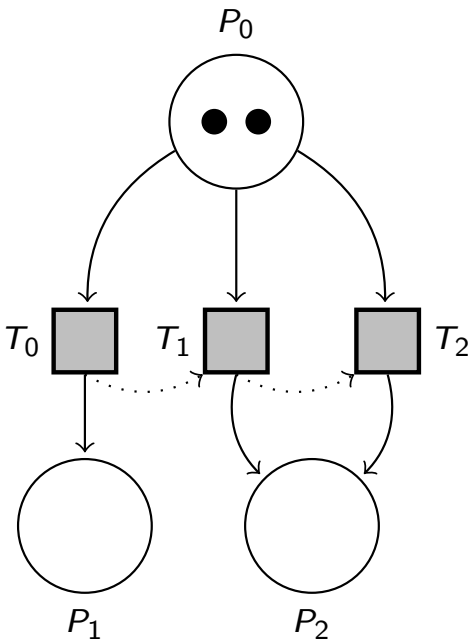



```
fired_transitions ← []

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.

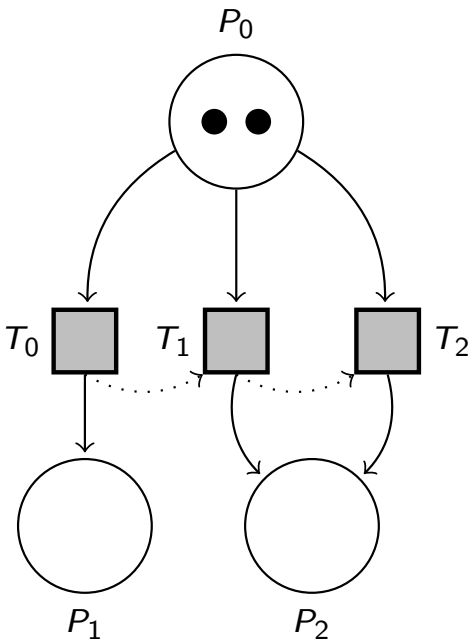



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

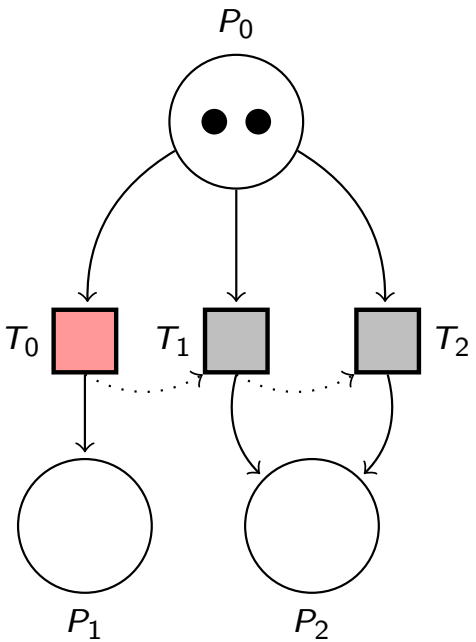



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

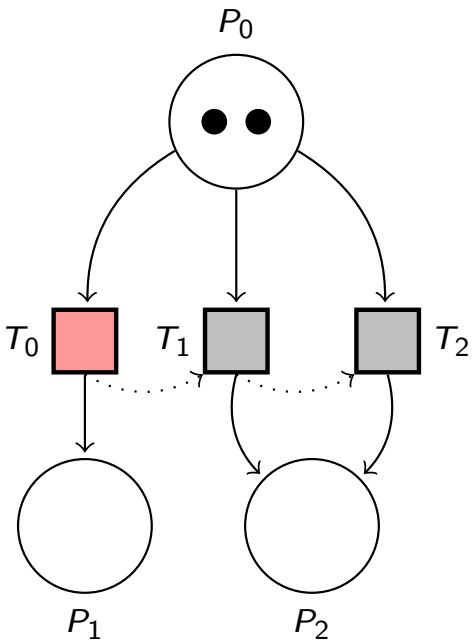



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

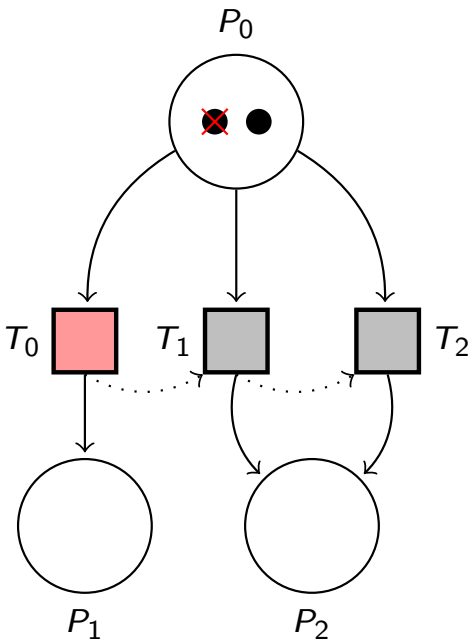



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

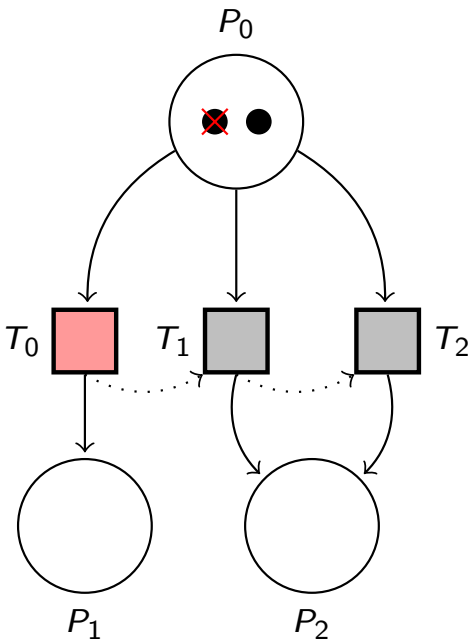



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

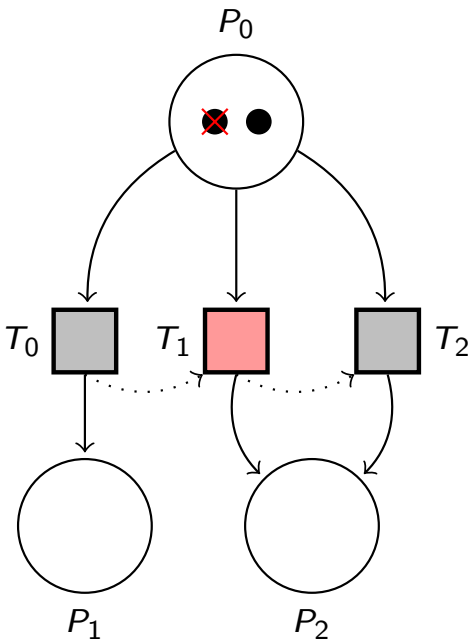



```
fired_transitions ← []

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.

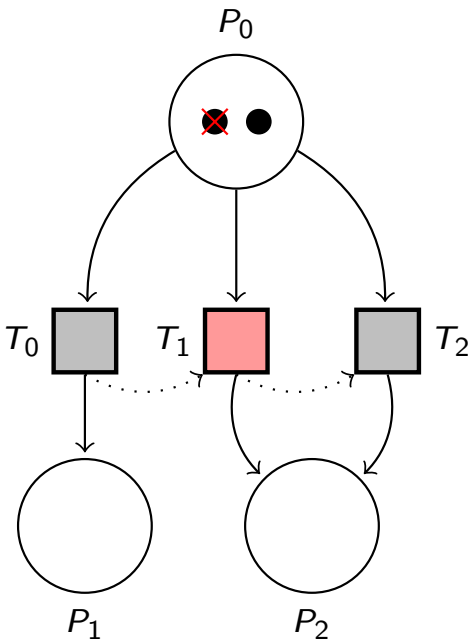



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

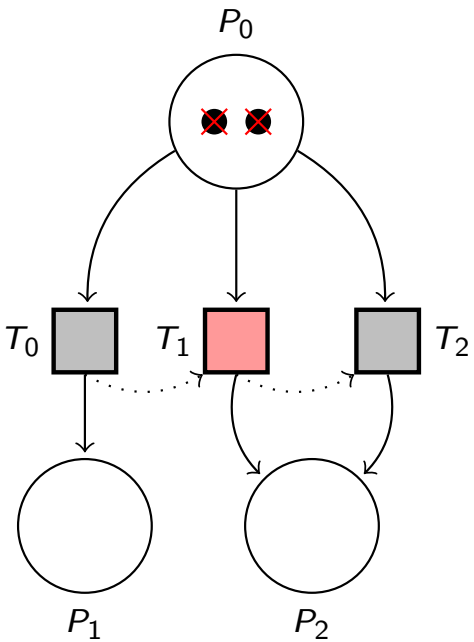



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

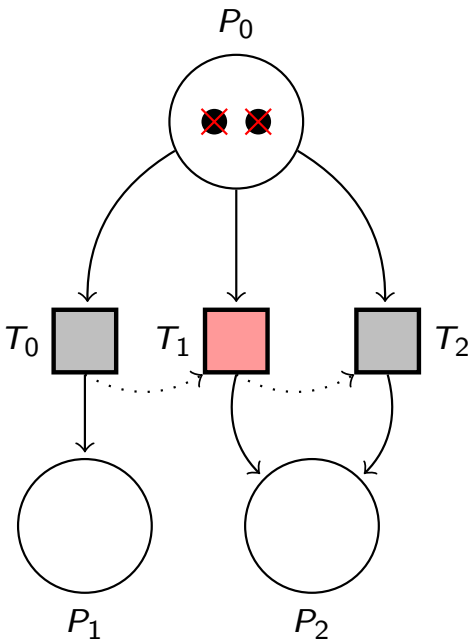



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

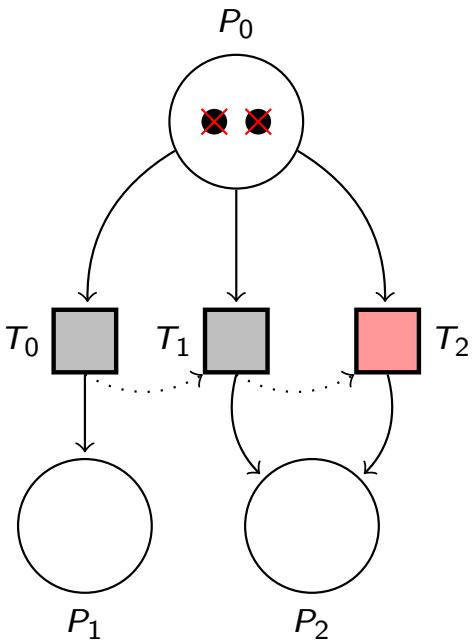



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

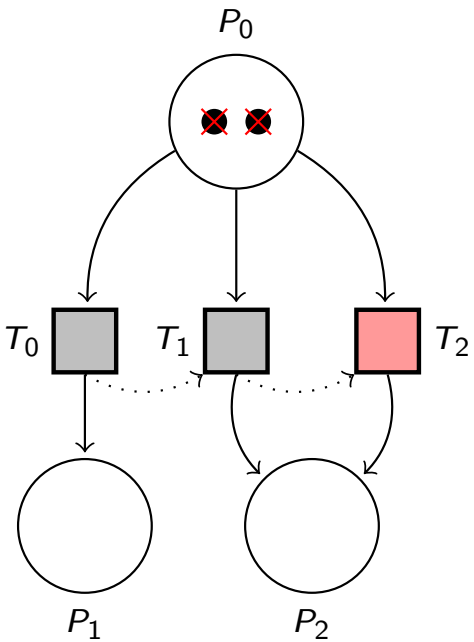



```
fired_transitions ← []

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.

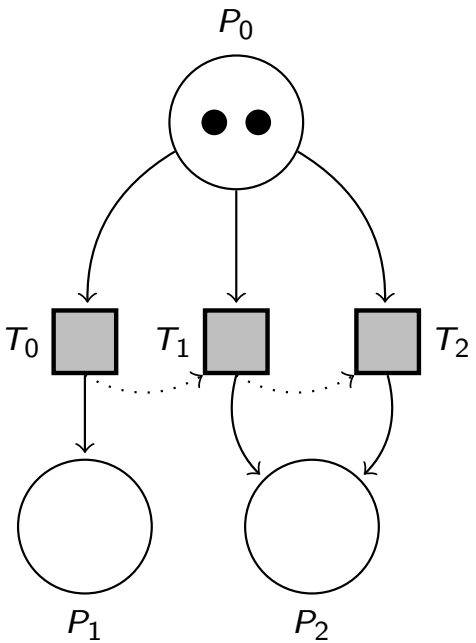



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
        then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

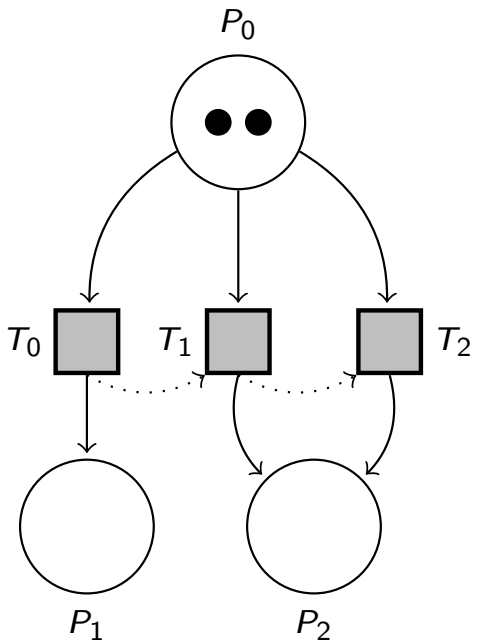



```
fired_transitions ← []
foreach priority_group in spn.priority_groups do
    resid_m ← s.marking
    foreach trans in priority_group do
        if is_firable(trans, s) and is_sensitized(trans, resid_m)
         then
            update_residual_marking(trans, resid_m)
            push_back(trans, fired_transitions)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

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

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)

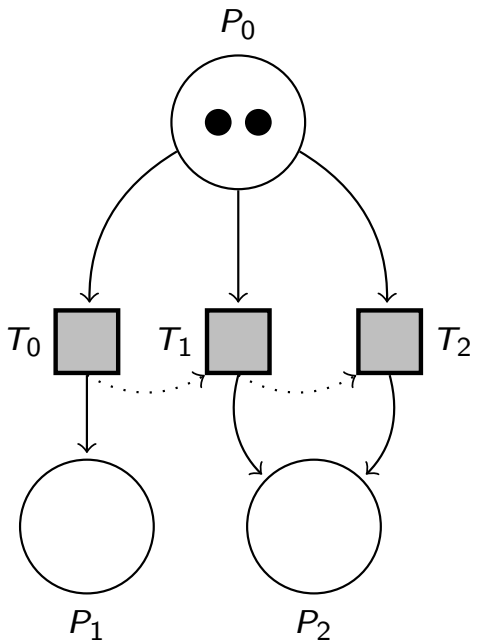
s'' ← make_state(s'.fired, new_marking)

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)

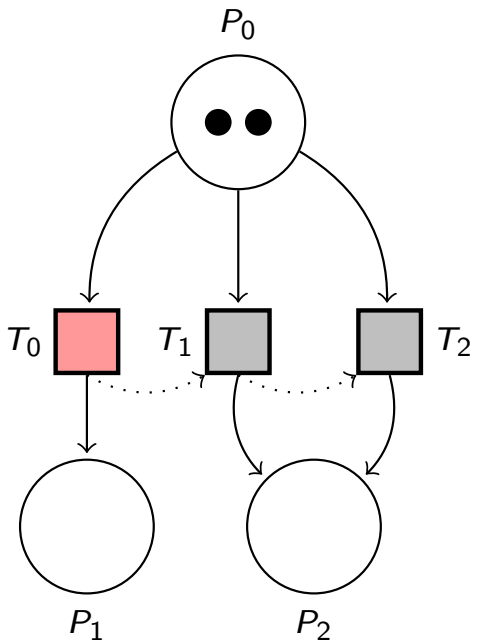
s'' ← make_state(s'.fired, new_marking)

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)

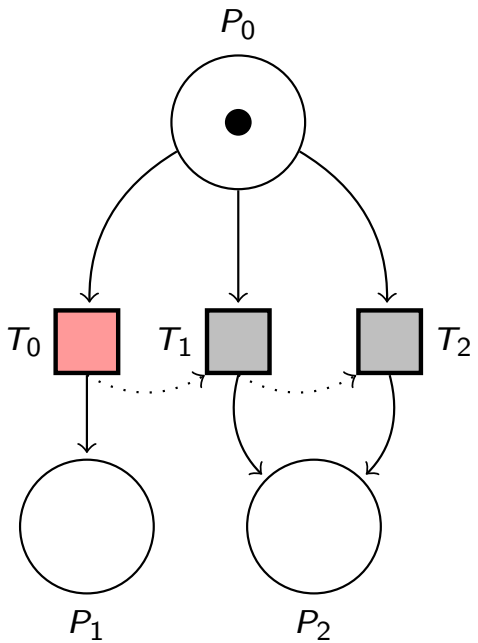
s'' ← make_state(s'.fired, new_marking)

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)

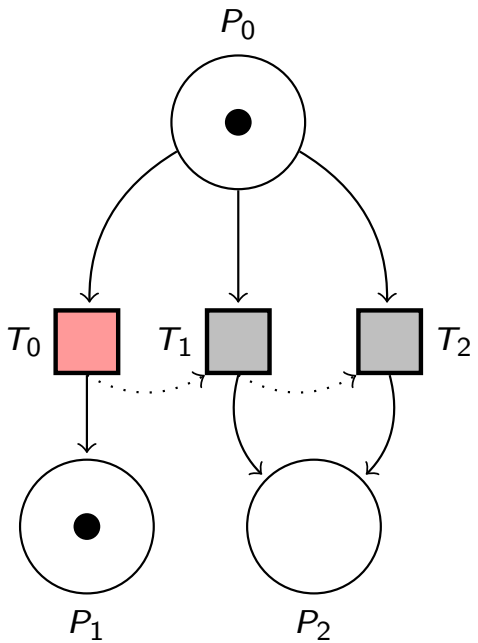
s'' ← make_state(s'.fired, new_marking)

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)

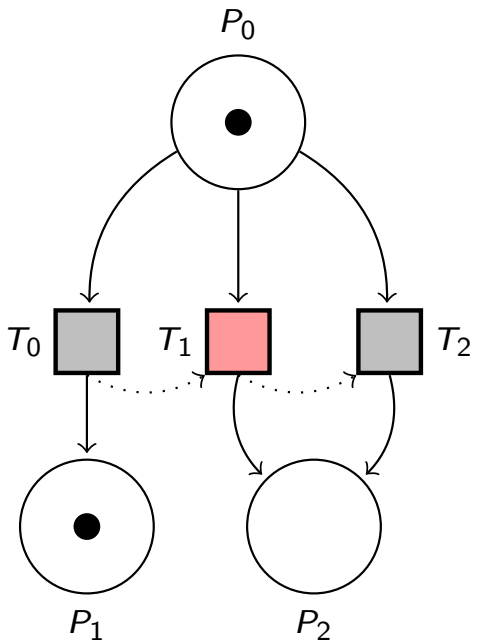
s'' ← make_state(s'.fired, new_marking)

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.

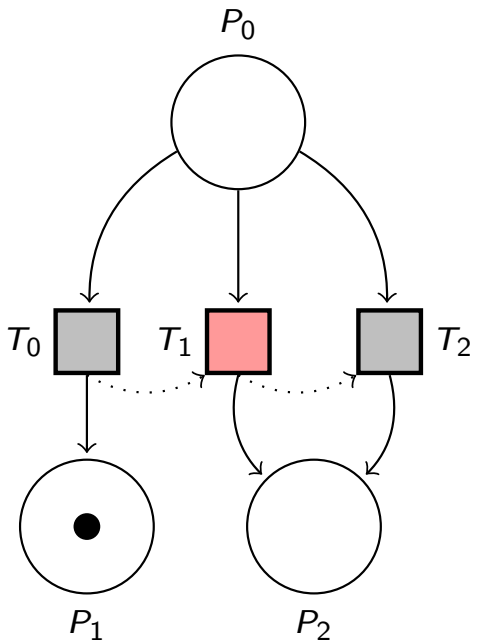



```
new_marking ← s'.marking
foreach trans in fired_transitions do
    update_marking_pre(trans, new_marking)
    update_marking_post(trans, new_marking)
s'' ← make_state(s'.fired, new_marking)
return (s', s'')
```

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

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)

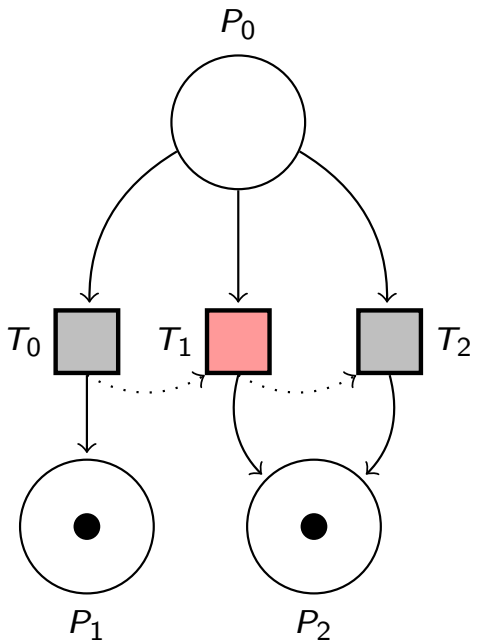
s'' ← make_state(s'.fired, new_marking)

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)

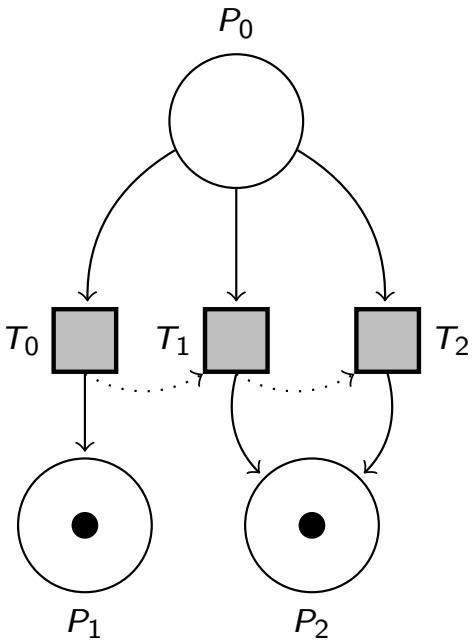
s'' ← make_state(s'.fired, new_marking)

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)

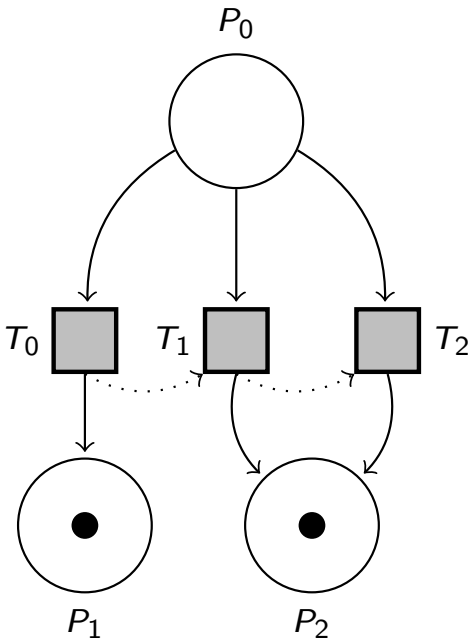
s'' ← make_state(s'.fired, new_marking)

return (s', s'')

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

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'')
```

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

Coq Implementation of the SPN Token Player.

```
1  Definition spn_cycle (spn : Spn) (starting_state : SpnState) :
2      option (SpnState * SpnState) :=
3      (* Computes the transitions to be fired. *)
4      match spn_falling_edge spn starting_state with
5      | Some inter_state ⇒
6        (* Updates the marking. *)
7        match spn_rising_edge spn inter_state with
8        | Some final_state ⇒ Some (inter_state, final_state)
9        | None ⇒ None
10       end
11     | None ⇒ None
12     end.
```

Figure: The SPN Token Player Program in Coq.

Coq Implementation of the SPN Token Player.

```
1  Definition spn_cycle (spn : Spn) (starting_state : SpnState) :
2      option (SpnState * SpnState) :=
3      (* Computes the transitions to be fired. *)
4      match spn_falling_edge spn starting_state with
5      | Some inter_state ⇒
6        (* Updates the marking. *)
7        match spn_rising_edge spn inter_state with
8        | Some final_state ⇒ Some (inter_state, final_state)
9        | None ⇒ None
10       end
11     | None ⇒ None
12     end.
```

Figure: The SPN Token Player Program in Coq.

- `match` checks the result of function calls.

Coq Implementation of the SPN Token Player.

```
1  Definition spn_cycle (spn : Spn) (starting_state : SpnState) :
2      option (SpnState * SpnState) :=
3      (* Computes the transitions to be fired. *)
4      match spn_falling_edge spn starting_state with
5      | Some inter_state ⇒
6        (* Updates the marking. *)
7        match spn_rising_edge spn inter_state with
8        | Some final_state ⇒ Some (inter_state, final_state)
9        | None ⇒ None
10       end
11     | None ⇒ None
12     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 $P \in X \to Y$ be a program and $S \in X \to Y \to \{\top, \bot\}$ be its specification.
P takes $x \in X$ as an input value and returns some $y \in Y$, S is a predicate that takes x and y as input values.

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/Completeness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*

$$\texttt{spn\_cycle spn s = Some (s', s'')} \Rightarrow s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s''.$$

Correctness/Completeness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*

$spn\_cycle\ spn\ s = Some\ (s',\ s'') \Rightarrow s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s''.$

Theorem (Completeness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*

$s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s'' \Rightarrow spn\_cycle\ spn\ s = Some\ (s',\ s'').$

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.

Done.

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

Conclusion.

Done.

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

Doing.

Add time, interpretation and macroplaces to SPNs semantics.

Conclusion.

Done.

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

Doing.

Add time, interpretation and macroplaces to SPNs semantics.

To Do.

Conclusion.

Done.

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

Doing.

Add time, interpretation and macroplaces to SPNs semantics.

To Do.

- Handle asynchronous communication in a synchronous execution paradigm (GALS).

Conclusion.

Done.

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

Doing.

Add time, interpretation and macroplaces to SPNs semantics.

To Do.

- Handle asynchronous communication in a synchronous execution paradigm (GALS).
- Model VHDL semantics (at least a subpart).

Conclusion.

Done.

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

Doing.

Add time, interpretation and macroplaces to SPNs semantics.

To Do.

- Handle asynchronous communication in a synchronous execution paradigm (GALS).
- Model VHDL semantics (at least a subpart).
- Implement the model-to-text transformation.

Conclusion.

Done.

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

Doing.

Add time, interpretation and macroplaces to SPNs semantics.

To Do.

- Handle asynchronous communication in a synchronous execution paradigm (GALS).
- Model VHDL semantics (at least a subpart).
- Implement the model-to-text transformation.
- Establish the proof of behavior preservation.

Bibliography.

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.


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


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

Medical Implants: An Application of HILECOP.

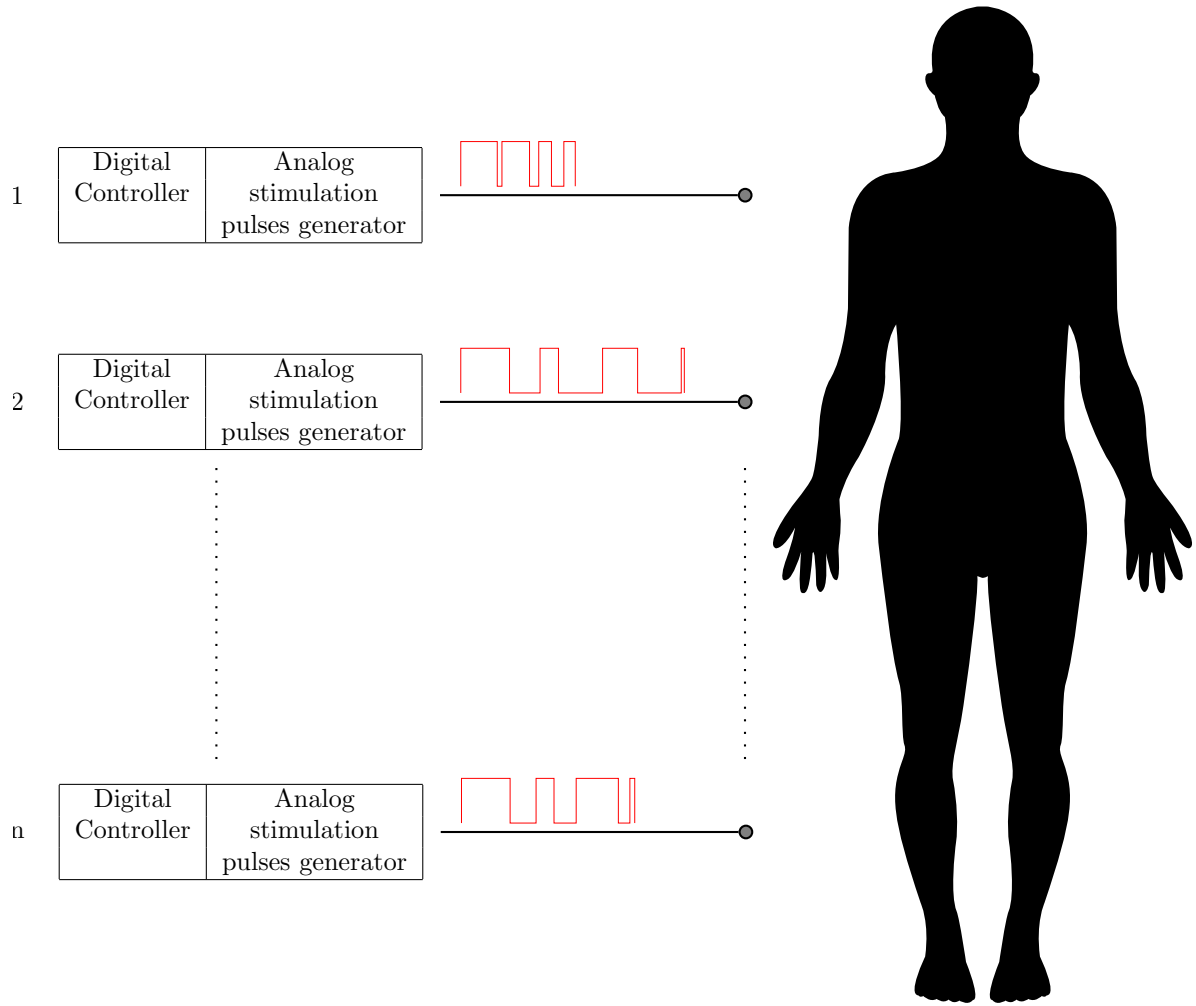


Figure: A Schematic Representation of NEURINNOV's Implantable Neuroprotheses.

HILECOP and CE Certification.

CE Certification for Medical Devices. [1]

- European Law on Medical Devices (2017/745).
- To obtain the certification:
  - Tests on devices (technologic, clinical).
  - Tests on elements of the production chain.

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

Correctness of The SPN Token Player.

Theorem (Correctness)

$\forall$ *(spn : Spn) (s s' s'' : SpnState), which are well-defined,*

$$\texttt{spn\_cycle spn s = Some (s', s'')} \Rightarrow s \overset{\downarrow clock}{\rightsquigarrow} s' \overset{\uparrow clock}{\rightsquigarrow} s''.$$

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,*

`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'`.

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'.`