Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 214 (2008) 231–254
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Experimenting Formal Proofs of Petri Nets Refinements

Christine Choppy, Micaela Mayero, Laure Petrucci [1]

Université Paris 13, LIPN, UMR CNRS 7030
99, av. J.-B. Clément
93430 Villetaneuse, FRANCE

Abstract

Petri nets are a formalism for modelling and validating critical systems. Generally, the approach to specification starts from an abstract view of the system under study. Once validated, a refinement step takes place, enhancing some parts of the initial model so as to obtain a more concrete specification. Some refinement techniques have been proposed in the framework of high-level Petri nets. Up to now, proving that a concrete net refines an abstract one, i.e. that there is a refinement relation between them, is completely manual. Our work aims at proving the refinement relation between two nets, both formally and automatically. For that purpose, we use the Coq theorem prover. We aim at having a framework general and parameterised enough to use Coq for any input nets. Moreover, this work constitutes a stepping stone towards bridging the gap between Petri nets and proof assistants techniques, and we claim that theorem proving methods are appropriate to prove the correctness of Petri net refinement.

Keywords: Petri net, theorem proving, refinement, Coq

1 Introduction

Modelling and analysing large and complex systems requires elaborate techniques and support. To overcome the problems inherent to designing and checking a large model (as well as the well-known state space explosion problem with model checking), a specification is often developed step by step. First, an abstract model is designed, and its properties are verified. Once it is proved correct, a refinement step takes place, introducing further detail. Such

[1] {Firstname.Lastname}@lipn.univ-paris13.fr,
http://www-lipn.univ-paris13.fr/~lastname/

an addition can either be enhancing the description of the actual functioning of part of the system, or introducing an additional part. This new refined model is then verified, and another refinement step can take place. This process is applied until an adequate level of description is obtained.

There are several advantages to use refinement and hence to start with a more abstract model. It gives a better and structured view of the system under study. The components within the system are clearly identified and the modeling process is eased. The modeller does not have to bother with spurious details. The validation process also becomes easier: the system properties are checked at each step. Thus, abstract models are validated before new details are added. Moreover, when model checking is used, the analysis of a full concrete model may not be amenable, due to its very large state space. Refinement helps in coping with this problem since it may preserve some properties or analysis results obtained at an earlier step for a more abstract model may be used for the analysis of the refined model. For example, Lakos and Lewis [10] use the state space of the abstract model to compute the state space of the refined one: some tests for enabledness of transitions are avoided, as well as the construction of partial markings that have already been computed. Moreover, the state space can be structured. Hence, this approach saves both space and time in the analysis, countering the state space explosion problem.

In this paper, we consider specifications written as coloured Petri nets [8]. They present the advantages of being quite compact, having a graphical representation, a formal semantics, and data handling. Three types of refinement have been defined for this kind of nets in [10]: type refinement, which addresses the refinement of datatypes handled ; node refinement, consisting in detailing the behaviour within a place or a transition ; and subnet refinement which allows for the addition of a connected subnet. Such refinements can be applied only if some constraints are satisfied. Up to now, these constraints must be proved manually, which is an error-prone process. Our aim is to formally prove that two nets are related by a refinement relation, using theorem proving techniques. While most Petri net properties are model checked (which involves traversing the state space), theorem proving (using proof assistants as B [1], PVS [12], Isabelle/HOL [11], Coq [5], . . .) is appropriate to check the Petri net refinement correctness. Let us note that, in this context, the goal of refinement is not to produce a specification close to the code, but rather to overcome the state space explosion problem through modularity.

The paper is structured as follows. Section 2 recalls the definitions of coloured Petri nets and the different refinements. Then, the deduction systems and their use are presented in section 3. Afterwards, a case study is addressed in section 4, where we describe different formalisations experimented. Conclusions and future work are finally discussed in section 5.

2 Petri Nets and Refinement

In this section, we present the formalisms used in this paper, and in particular the kind of Petri nets we consider. We also describe the various Petri nets refinements, briefly for the type refinement, and with more details for the subnet and node refinements. We do not describe here the formal specification regarding type refinement, but we give a sufficient description for the reader to understand the discussion in section 5.

We end this section with correctness lemmas for refinement. These lemmas will be formalised and proved with the Coq theorem prover in section 4.

2.1 Basic Definitions

The Petri nets refinements we base on, from [9], were defined for Coloured Petri Nets (CPN). The definition of refinement is first adapted here for place/transition Petri nets. We will show our approach first for these and will extend the work by adding colours in section 4.5.

Definition 2.1 [Petri Net] A Petri Net $\mathcal{N}$ is a 4-tuple $\mathcal{N} = \langle P, T, W, M_0 \rangle$ where:

a. P is a set of *places*

b. T is a set of *transitions*, s.t. $P \cap T = \emptyset$

c. $W : (P \times T) \cup (T \times P) \longrightarrow \mathbb{N}$ is the valuation function of *arcs* ;

d. $M_0 : P \longrightarrow \mathbb{N}$ is the *initial marking*.

A Petri net can be represented as a bipartite graph. Note that there is at most one arc in each direction for a given couple (place,transition), and the action of an arc is expressed by its direction and the value of W. Note that if W is null there is no arc.

Places may contain any number of tokens. A distribution of tokens over the places of a net is called a marking. The initial marking is a particular case which expresses the initial state of the system (see figure 1 for an example).

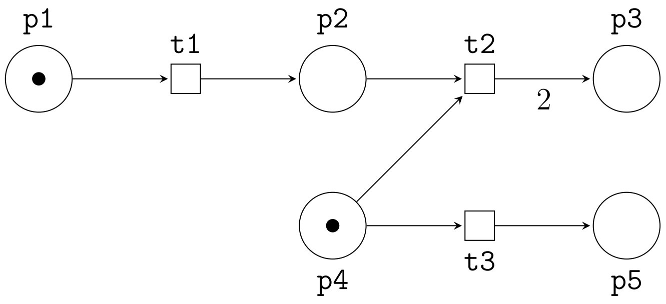

Fig. 1. Initial net

Definition 2.2 [Marking] Let $\mathcal{N} = \langle P, T, W, M_0 \rangle$ be a Petri net. A *marking* of $\mathcal{N}$ is a function $M : P \longrightarrow \mathbb{N}$.

When a system is in a state M, some transitions can be fired and we have a new state M'.

Definition 2.3 [Behaviour] Let $\mathcal{N} = \langle P, T, W, M_0 \rangle$ be a Petri net and M a marking of $\mathcal{N}$. A transition $t \in T$ is *enabled* in the marking M (noted $M[t\rangle$) iff :

$$\forall p \in P : W(p,t) \leq M(p)$$

In this case, the *firing* of the transition t in the marking M leads to a new marking M' (denoted by $M[t\rangle M'$) such that :

$$\forall p \in P : M'(p) = M(p) - W(p,t) + W(t,p)$$

The behaviour of a Petri net is the set of possible transitions firing sequences from the initial marking.

Example: from the initial marking in figure 1, transition `t1` is enabled and its firing leads to the marking with one token in `p2` and `p4`, and none in the other places.

As mentioned above, three types of refinement have been defined. For the reader to understand the discussion in section 5, we provide below a brief description of these refinements. A mathematical description of the subnet refinement will also be given, since it will be formalised in section 4. The three types of refinement are presented here, and further described in [9].

The refinements should preserve the Petri net behaviour, i.e. it should always be possible to associate a behaviour of the refined net with a corresponding behaviour of the abstract one for those transitions that are in both nets.

2.2 Type Refinement

In a coloured net, information is carried by the tokens.

Type refinement is related to the information carried by the tokens (as well as in the firing modes), while the net structure remains unchanged. Type refinement involves incorporating additional information. This can be done in various ways, for instance by adding a new element in a tuple, or by representing an abstract type by a more concrete (detailed) one. Type refinement should preserve the type properties, i.e. if type A is refined by type B, the properties satisfied by A should also be satisfied by B (after an adequate syntactical translation).

In section 4.5, we formalise (a subclass of) coloured nets and discuss in section 5 the formalisation of asociated proofs of refinement properties.

2.3 Node Refinement

The formal definition of this refinement is given in [9]. As for type refinement (see Section 2.2), we will only give an idea of that kind of refinement. This refinement has been formalised using one of the methods explained in section 4; we do not give all details here, but we will refer to them in section 5.

Node refinement consists in replacing a place (transition) by a place- (transition-) bordered subnet. This refinement consists in canonical place and transition refinements.

The following definition expresses a place refinement of the abstract net, where P_{in}, P_{out} and P' are sets of new places.

Definition 2.4 [Place Refinement] Let $\mathcal{N}_r = \langle P_r, T_r, W_r, M_{0r}\rangle$ and $\mathcal{N}_a = \langle P_a, T_a, W_a, M_{0a}\rangle$ be two Petri nets. $\mathcal{N}_r$ is a *place refinement* of net $\mathcal{N}_a$ w.r.t. p_r if :

(i) $\forall p \in P_a \setminus \{p_r\}, \forall t \in T_a : p \in P_r \wedge t \in T_r \wedge W_r(p,t) = W_a(p,t) \wedge W_r(t,p) = W_a(t,p) \wedge M_{0r}(p) = M_{0a}(p)$;

(ii) $P_r = P_{in} \cup P_{out} \cup P' \cup P_a \setminus \{p_r\}$ where $\forall p \in P' : M_{0r}(p) = 0$, $\forall t \in T_a, \forall p \in P_{in} : W_r(p,t) = 0 \wedge W_r(t,p) = W_a(t,p_r)$, and $\forall t \in T_a, \forall p \in P_{out} : W_r(t,p) = 0 \wedge W_r(p,t) = W_a(p_r,t)$;

(iii) The token flow through the place refinement border is preserved w.r.t. the abstract net;

(iv) $\forall p \in P_r \setminus P_a, \forall t \in T_r \setminus T_a : W_r(p,t) \in \{0,1\} \wedge W_r(t,p) \in \{0,1\}$.

The following definition is similar, for transition refinement.

Definition 2.5 [Transition Refinement] Let $\mathcal{N}_r = \langle P_r, T_r, W_r, M_{0r} \rangle$ and $\mathcal{N}_a = \langle P_a, T_a, W_a, M_{0a} \rangle$ be two Petri nets. $\mathcal{N}_r$ is a *transition refinement* of net $\mathcal{N}_a$ w.r.t. t_r if :

(i) $\forall p \in P_a, \forall t \in T_a \setminus \{t_r\} : p \in P_r \wedge t \in T_r \wedge W_r(p,t) = W_a(p,t) \wedge W_r(t,p) = W_a(t,p) \wedge M_{0r}(p) = M_{0a}(p)$;

(ii) $T_r = T_{in} \cup T_{out} \cup T' \cup T_a \setminus \{t_r\}$ where $\forall p \in P_a, \forall t \in T_{in} : W_r(t,p) = 0 \wedge W_r(p,t) = W_a(p,t_r)$, and $\forall p \in P_a, \forall t \in T_{out} : W_r(p,t) = 0 \wedge W_r(t,p) = W_a(t_r,p)$;

(iii) The token flow through the transitions refinement border is preserved w.r.t. the abstract net;

(iv) $\forall p \in P_r \setminus P_a, \forall t \in T_r \setminus T_a : W_r(p,t) \in \{0,1\} \wedge W_r(t,p) \in \{0,1\}$;

(v) $\forall p \in P_r \setminus P_a : M_{0r}(p) = 0$.

2.4 Subnet Refinement

As mentioned previously, this kind of refinement is the main one considered in this paper, as will be illustrated by the case study.

Subnet refinement consists in adding net components (places, transitions and arcs or even additional tokens or mode values).

Definition 2.6 [Subnet Refinement] Let $\mathcal{N}_r = \langle P_r, T_r, W_r, M_{0r} \rangle$ and $\mathcal{N}_a = \langle P_a, T_a, W_a, M_{0a} \rangle$ be two Petri nets. $\mathcal{N}_r$ is a *subnet refinement* of $\mathcal{N}_a$ if :

(i) $P_a \subseteq P_r$;

(ii) $T_a \subseteq T_r$;

(iii) $\forall p \in P_a, \forall t \in T_a : W_r(p,t) = W_a(p,t) \wedge W_r(t,p) = W_a(t,p)$
$\forall p \in P_a, \forall t \in T_r \setminus T_a : W_r(p,t) = W_r(t,p) = 0$
$\forall p \in P_r \setminus P_a, \forall t \in T_a : W_r(t,p) = 0$;

(iv) $\forall p \in P_a : M_{0r}(p) = M_{0a}(p)$.

The subnet refinement can be seen as an extension of initial abstract net with a subnet linked only by input arcs to abstract net transitions. This ensures that the behaviour preservation condition holds. Indeed, when adding input to transitions the firing is more constrained. Therefore, if the transition is enabled in the refined net, it is also enabled in the abstract one.

For example, subnet refinement can be applied to the net of figure 1 to obtain the net of figure 2, where places `p6`, `p7` and transition `t4` have been added.

In the refined net of figure 2 the firable sequences are `t1 t2` *t4*, *t4* `t1 t2`,

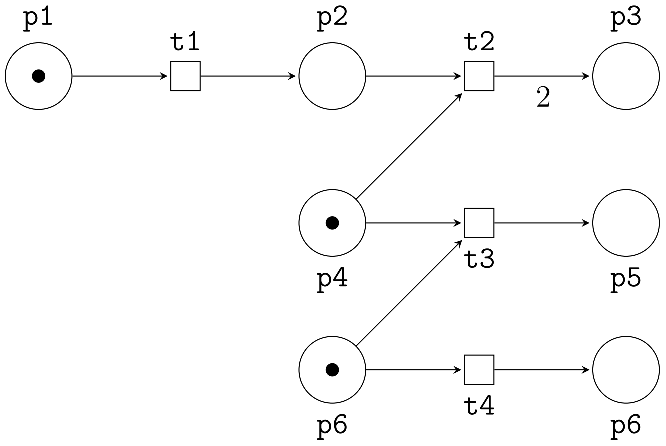

Fig. 2. Subnet refinement

`t1` *t4* `t2`, `t1 t3`, `t3 t1`. Note that as soon `t4` is fired, `t3` will not be enabled anymore.

Firable sequences in the abstract net of figure 1 are `t1 t2, t1 t3` and `t3 t1`.

The following soundness lemma for subnet refinement expresses definition 2.6 in a way that can be translated and proved using Coq (see the formalisation in section 4).

Lemma 2.7 *Given P', T' and A', the sets of places, transitions and arcs of the refined net, and P, T, A, the set of places, transitions and arcs of the initial (abstract) net, the subnet refinement satisfies the following properties:*

(i) $P \subseteq P'$

(ii) $T \subseteq T'$

(iii) $A \subseteq A'$

(iv) *There is no new arc from P to T'.*

(v) *There is no new arc from T to P'.*

(vi) *There is no new arc from T' to P.*

(vii) *The initial marking of places in P is unchanged.*

3 Using a Deduction System: Coq System presentation

Automated theorem proving consists in proving mathematical theorems using a computer program. Interactive theorem provers require a user to give hints to the system. Depending on the degree of automation, the prover can essentially be reduced to a proof checker, with the user providing the proof in a formal way, or significant proof tasks can be performed automatically. We have chosen Coq in order to specify and prove our refinement problem.

The Coq [5] tool is a formal proof assistant system: a proof performed using Coq is mechanically checked by the machine. In particular, Coq allows for defining functions or predicates, stating mathematical theorems and software specifications, developing interactively formal proofs of these theorems, and checking these proofs by a small certification "kernel".

Coq is based on a logical framework called "Calculus of Inductive Constructions" [13] extended by a modular development system for theories.

In order to help the reader who may not be familiar with this kind of tool, we first present three small examples which may be useful to understand our formalisations. In particular we present three different methods (datatypes) that can be used in Coq to formalise sets, and that we use in Section 4 for the Petri nets encoding.

3.1 Sets library

In this case study formalisation of section 4.2, we use the `Sets` library that comes with the Coq system distribution.

Let us first get acquainted with Coq and the `Sets` library with a few examples.

In the standard library, sets (called `Ensembles`) are defined as follows:

```
Variable U : Type.
Definition Ensemble := U -> Prop.
```

`U` is the type of the set elements, and it is a parameter. `Prop` is the type of propositions of Coq. Thus, the set of integers `E` is defined by `E:(Ensemble nat).`

We now define the membership and union concepts:

```
Definition In (A:Ensemble) (x:U) : Prop := A x.

Inductive Union (B C:Ensemble) : Ensemble :=
  | Union_introl : forall x:U, In B x -> In (Union B C) x
  | Union_intror : forall x:U, In C x -> In (Union B C) x.
```

The following table presents some correspondences between the usual mathematical notations (left column) and their expression in Coq (right column), where A and B are integer sets:

$x \in A$	`In nat A x`
$A \cup B$	`Union nat A B`
$B \setminus A$	`Setminus nat A B`

We can, for instance, prove the property: if $x \in B \cup C$ then $x \in B$ or $x \in C$, that is expressed and proved with Coq as follows.

```
Lemma Union_inv :
 forall (B C:Ensemble U) (x:U), In U (Union U B C) x ->
        In U B x \/ In U C x.
Proof.
intros B C x H'; elim H'; auto with sets.
Qed.
```

The tactic `intros` allows us to consider the expressions on the left-hand side of the arrows (here, `B, C, x` and `(In U (Union U B C) x)`) as hypotheses, the tactic `elim` allows us to apply an induction scheme (corresponding to an inductive type, here `Union`). The tactic `auto` is an automation tactic (trying to apply lemmas stored in a database).

3.2 *`FSets` library*

In the case study formalisation of section 4.3, we use the `FSets` library that comes with the Coq system distribution. It is a finite sets library. It actually contains three different implementations of finite sets using the Coq functors (parameterised modules) and modules: ordered lists, red-black trees and AVLs.

In the example below, we will define two sets A and B, the first one contains integers 1, -2 and 3 and the second one contains 4. Next, we can prove that 1 belongs to the union of A and B.

We first import the `FSets` library, next the `ZArith` library (of integers) and finally open the "scope" of Z which allows us to use the integers syntax:

```
Require Import ZArith FSets.
Open Scope Z_scope.
```

We can now define a new module from the functor `FSetList`. To do that, we have to pass a module representing the integers seen as ordered type as argument; it is the `Z_as_OT` module which is available in the library. Our integer set will be the following `SetZ`:

```
Module SetZ := FSetList.Make Z_as_OT.
```

To define A and B, we can use functions `add` and `empty` defined in the implementation by ordered lists. For example, for `add`, it is an insertion sort (doubles are ignored); `empty` is the empty list. We can define $A = \{1, -2, 3\}$ and $B = \{4\}$ as follows:

```
Definition A := SetZ.add 1 (SetZ.add (-2) (SetZ.add 3 SetZ.empty)).
Definition B := SetZ.add 4 SetZ.empty.
```

We can now prove that $1 \in A \cup B$. To do that, the user has to apply 2 lemmas of the library: `union_2` and `add_1`. `union_2` states that if s and s' are 2 sorted lists, if x and y are elements, and if x is in the set s then x is also in the union of s and s'. `add_1` states that if x and y are equal then y is in s where we have added x. In this proof, we also have to expand A by $\{1, -2, 3\}$, B by $\{4\}$ using Coq command `unfold`. Finally, we obtain the following Coq script:

```
Lemma prop1: SetZ.In 1 (SetZ.union A B).
Proof.
 apply SetZ.union_2.
 unfold A.
 apply SetZ.add_1.
 auto.
Qed.
```

where the `auto` command is needed to prove $1 = 1$ and `union_2` and `add_1` are defined as follows:

```
Lemma union_2 : forall (s s': t)(Hs: Sort s)(Hs': Sort s')(x: elt),
   In x s -> In x (union s s').
Lemma add_1 : forall (s: t)(Hs: Sort s)(x y: elt), X.eq x y ->
   In y (add x s).
```

Note that there may exist many ways to make a proof. Another possible proof is the following:

```
Lemma prop1: SetZ.In 1 (SetZ.union A B).
Proof.
 Apply SetZ.mem_2; unfold SetZ.mem; simpl; auto.
Qed.
```

This second proof presents the advantage of simplicity and efficiency[2]. This is interesting to write automated tactics.

3.3 List library and Record type

In the case study formalisation of section 4.4, we use the `List` library that comes with the Coq system distribution. We represent sets as lists. We first import the `List` library and open the ”scope” of nat and list which allow us to use natural numbers and list syntax respectively:

```
Require Export List.
Open Scope nat_scope.
Open Scope list_scope.
```

The type `list` is defined as an inductive type with two constructors `nil` and `cons` (denoted by `::`):

```
Inductive list (A : Type) : Type :=
     nil  : list A
   | cons : A -> list A -> list A
```

Now, we are able to define a list which contains 3 natural numbers, 1, 2 and 3:

```
Definition example_list_nat:=(1::2::3::nil).
```

Then, the type of `example_list_nat` is `list nat`. The standard library of lists includes many properties that allows us to make proofs using lists for our specification.

Now, we also present the record type that will be used in the section 4. The record construction is a macro allowing the definition of records as is done in many programming languages. Actually, the macro generates an inductive definition with just one constructor. As an example, we can define complex numbers, with real and imaginary parts (we have to require the Real library first) and $2 + i$:

```
Require Export Reals.
Open Scope R_scope.

Record C : Set := mkC
```

[2] ”Simplicity” means facility to manipulate objects and data types in proofs, in other words, facility to make proofs. ”Efficiency” means size of proof λ-term (to reduce time of type checking), using conversion (by reflexion) instead of rewriting for example. For further details, the reader can refer to [5].

```
  {real : R;
   im : R}.

Definition two_plus_i:=mkC 2 1.
```

For further explanation about the previous examples of formalisations, refer to [5].

4 Proving Refinement Using Coq

To formally prove a refinement we need to prove, with Coq, the correctness properties expressed in section 2. In this section we sketch two possible ways to achieve this proof.

(i) The first method is, as in the previous section, to manually drive the proof, using existing tactics in the Coq prover. Of course this is much simpler, but the drawback is that it cannot be generalised since it is valid for each specific proof and cannot be reused.

(ii) The second approach is to build the proof such that it is as automated as possible. To automate a proof means that our specific tactics should be explicited and written in such a way that they may be applied to most of our problems.
With Coq, the user may write her/his own automation tactics thanks to the tactic language $\mathcal{L}_{tac}$ [7]. In this section, we present as an example a short tactic that may be used to automate our case study proofs, and may also be applied in a more general context to coloured Petri nets.

4.1 Case study description

As a case study, we chose the quite simple place transition net derived from [9]. The initial net is shown in figure 1.

Subnet refinement involves augmenting a subnet with additional places, transitions, and arcs. Subnet refinement can be applied to the net of figure 1 to obtain the net of figure 2, where places `p6`, `p7` and transition `t4` have been added.

We present hereafter three main parts of formalisations of the subnet refinement formal proof (lemma 2.7) in commented Coq code that use the three sets formalisations presented in Section 3. Note that the properties introduced by these different representations are not involved in our proofs, so the difference between the three possibilities will only be in the way the proof can be achieved, and also generalised.

4.2 Using the `Sets` library

As described in section 3.1, we import the required libraries:

```
Require Export Ensembles.
```

4.2.1 Formalisation of a place/transition net

Defining places and transitions is achieved inductively. They are indexed by natural numbers.

```
Inductive places : Type := p : nat -> places.

Inductive transitions : Type := t : nat -> transitions.
```

We distinguish arcs from a place to a transition and arcs from a transition to a place. Arcs are labelled by their weight (for simple Petri nets, the weight is the number of tokens involved in firing the transition). An arc from a place to a transition can be represented as a pair ((place, transition),n) where n is its label (that is the weight of the arc), and similarly ((transition, place),n) for an arc from a transition to a place. A marking can be represented as a pair (place,n) where n is the number of tokens.

```
Definition arc_pt (pl:places)(tr:transitions)(n: nat):=
  (pairT (pairT pl tr) n).

Definition arc_tp (tr:transitions) (pl:places)(n: nat):=
  (pairT (pairT tr pl) n).

Definition marking (pl:places)(n:nat):=(pairT pl n).
```

It is now possible to define the net in figure 1. The set `P` containing places `Pj, j=1..5` in the initial net is defined as:

```
Definition P1 := p 1.
...
Definition P5 := p 5.

Definition P :=
  Add places (Add places (Add places (Add places (Add places
    (Empty_set places) P1) P2) P3) P4) P5.
```

The three transitions are defined in a similar manner:

```
Definition T1 := t 1.
...
```

```
Definition T :=
  Add transitions (Add transitions (Add transitions
        (Empty_set transitions) T1) T2) T3.
```

Apt and Atp respectively represent the set of arcs from places to transitions, and the set of arcs from transitions to places. Note that the definition of the arc At2p3 from transition t_2 to place p_3 is labelled with 2.

```
Definition Ap1t1 := arc_pt P1 T1 1.
...
Definition Ap4t3 := arc_pt P4 T3 1.
Definition At1p2 := arc_tp T1 P2 1.
...

Definition Apt :=
  Add (prodT (prodT places transitions) nat)
  (Add (prodT (prodT places transitions) nat)
  (Add (prodT (prodT places transitions) nat)
    (Add (prodT (prodT places transitions) nat)
    (Empty_set (prodT (prodT places transitions) nat))
         Ap1t1) Ap2t2) Ap4t2) Ap4t3.
```

Atp is defined similarly from transitions to places.

The initial marking is the following:

```
Definition mP1:=marking P1 1.
Definition mP2:=marking P2 0.
Definition mP3:=marking P3 0.
Definition mP4:=marking P4 1.
Definition mP5:=marking P5 0.
```

```
Definition InitMarkI := Add (prodT places nat) (Add (prodT places nat)
  (Add (prodT places nat) (Add (prodT places nat) (Add (prodT places nat)
     (Empty_set (prodT places nat)) mP1) mP2) mP3) mP4) mP5.
```

4.2.2 Subnet refinement

The refined net is defined in Coq in a similar way. We denote respectively by P', T', Apt', Atp' and InitMarkR the sets of places, transitions, arcs (from places to transitions and conversely) and the initial marking.

```
Definition P'1 := p 1.
```

```
....
Definition P' :=
  Add places (Add places (Add places (Add places (Add places
              (Add places (Add places (Empty_set places)
                          P'1) P'2) P'3) P'4) P'5) P'6) P'7.

Definition T'1 := t 1.
....
Definition T' :=
  Add transitions (Add transitions (Add transitions
                   (Add transitions (Empty_set transitions)
                                    T'1) T'2) T'3) T'4.

Definition A'p1t1 := arc_pt P'1 T'1 1.
...
Definition A't1p2 := arc_tp T'1 P'2 1.
Definition A't2p3 := arc_tp T'2 P'3 2.
...

Definition A'pt :=
  Add (prodT (prodT places transitions) nat)
   (Add (prodT (prodT places transitions) nat)
     (Add (prodT (prodT places transitions) nat)
       (Add (prodT (prodT places transitions) nat)
         (Add (prodT (prodT places transitions) nat)
           (Add (prodT (prodT places transitions) nat)
             (Empty_set (prodT (prodT places transitions) nat))
              A'p1t1) A'p2t2) A'p4t2) A'p4t3) A'p6t3) A'p6t4.
```

`A'tp` is defined similarly from transitions to places.

The initial marking is defined by:

```
Definition mP'1:=marking P'1 1.
...
Definition mP'6:=marking P'6 1.
Definition mP'7:=marking P'7 0.

Definition InitMarkR := Add (prodT places nat) ...
  (Empty_set (prodT places nat)) mP'1) mP'2)  mP'3) mP'4) mP'5) mP'6) mP'7.
```

Now we can express the correctness lemma in Coq that corresponds to lemma 2.7. We do not give here the sequence of tactics that constitutes the proof.

```
Lemma subnet_refined: Included places P P' /\ Included transitions T T'
  /\ Included (prodT places nat) InitMarkI InitMarkR /\
  (Included  (prodT (prodT places transitions) nat) Apt A'pt /\
   (forall (pe:places) (te:transitions) (n:nat), (In places P pe) ->
   (In transitions T te) -> ~In (prodT (prodT places transitions) nat)
       (Setminus (prodT (prodT places transitions) nat) Apt A'pt)
          (pairT (pairT pe te) n))) /\
  (Included  (prodT (prodT transitions places) nat) Atp A'tp /\
  (forall (te:transitions) (pe:places) (n:nat), (In places P pe) ->
   (In transitions T' te) -> ~In (prodT (prodT transitions places) nat)
       (Setminus (prodT (prodT transitions places) nat) Atp A'tp)
          (pairT (pairT te pe) n))).
```

The Coq proof requires about 30 tactics that are specific to the previous proof. As mentioned before, one of our goals is to automate the proofs. To this end, we wrote a tactic that may be applied to any Petri net encoded through sets. We present part of the code below:

```
Ltac is_subnet_refinement :=
  try
   match goal with
   | |- (?X1 /\ ?X2) =>
       split;[try (match goal with | |- (Included _ ?X3 ?X4) =>
          unfold X3, X4; auto with sets end) | intros;
     (match goal with | |- ~(In _ (Setminus _ ?X3 ?X4) _ )=>
          unfold X3, X4 end);unfold Setminus;intro; ...]
   end.
```

The keyword `Ltac` means that we define a new tactic in Coq (called `is_subnet_refinement`), thanks to the tactic language $\mathcal{L}_{tac}$ mentioned in section 3. The keywords `match goal with` allow us to match the goal (i.e. hypothesis and conclusion) with some patterns and to apply the corresponding appropriate tactics (`split, unfold, ...`) to this goal.
The 30 tactics of the previous proof are therefore replaced by this unique and generic tactic `is_subnet_refinement`.

4.3 Using the FSets library

As described in section 3.2, we import the required libraries:

`Require Export FSets.`

4.3.1 Formalisation of a place transition net

We create, for places and transitions, a new module `PT` with `Nat_as_OT`, which represents a finite set of natural numbers:

`Module PT := FSetList.Make Nat_as_OT.`

As previously, arcs and marking are represented respectively by a triple (pl,tr,n) or (tr,pl,n) and by a pair (pl, n). In Coq, we define a triple (a,b,c) as a couple ((a,b),c), using predefined module `PairOrderedType`:

```
Module PairOT := PairOrderedType Nat_as_OT Nat_as_OT.
Module TriOT := PairOrderedType PairOT Nat_as_OT.
Module Arc := FSetList.Make TriOT.
Module Mark := FSetList.Make PairOT.
```

After that, the arcs and marking are defined:

```
Definition arc_pt (pl tr n: nat):= (pl,tr,n).
Definition arc_tp (tr pl n: nat):= (tr,pl,n).

Definition marking (pl n:nat):=(pl,n).
```

We can now define the abstract net of figure 1. Set `P` represents the set of places, from 1 to 5:

```
Definition P1 := 1.
...

Definition P := PT.add P1 (PT.add P2 (PT.add P3
    (PT.add P4 (PT.add P5 PT.empty)))).
```

The three transitions are defined similarly:

```
Definition T1 := 1.
...

Definition T := PT.add T1 (PT.add T2 (PT.add T3 PT.empty)).
```

`Apt` and `Atp` represent the sets of arcs from places to transitions and from transitions to places respectively. Note that the arc `At2p3` from transition t_2 to place p_3 is weighted by 2.

```
Definition Ap1t1 := arc_pt P1 T1 1.
...
Definition At1p2 := arc_tp T1 P2 1.
...

Definition Apt := Arc.add Ap1t1 (Arc.add Ap2t2
  (Arc.add Ap4t2 (Arc.add Ap4t3 Arc.empty))).

Definition Atp :=
  Arc.add At1p2 (Arc.add At2p3 (Arc.add At3p5 Arc.empty)).
```

The initial marking is represented as follows:

```
Definition mP1:=marking P1 1.
...
Definition InitMarkI := Mark.add mP1 (Mark.add mP2 (Mark.add mP3
   (Mark.add mP4 (Mark.add mP5 Mark.empty)))).
```

4.3.2 Subnet refinement

The refined net is defined similarly in Coq. We denote respectively by `P'`, `T'`, `Apt'`, `Atp'` and `InitMarkR` the sets of places, transitions, arcs and the initial marking.

```
Definition P'1 := 1.
....
Definition P' := PT.add P'1 (PT.add P'2 (PT.add P'3 (PT.add P'4
  (PT.add P'5 (PT.add P'6 (PT.add P'7 PT.empty)))))).

Definition T'1 := 1.
....
Definition T' := PT.add T'1 (PT.add T'2
  (PT.add T'3 (PT.add T'4 PT.empty))).

Definition A'p1t1 := arc_pt P'1 T'1 1.
...
Definition A't1p2 := arc_tp T'1 P'2 1.
Definition A't2p3 := arc_tp T'2 P'3 2.
...
Definition A'pt :=
  Arc.add A'p1t1 (Arc.add A'p2t2 (Arc.add A'p4t2 (Arc.add A'p4t3
    (Arc.add A'p6t3 (Arc.add A'p6t4 Arc.empty))))).

Definition A'tp :=
  Arc.add A't1p2 (Arc.add A't2p3 (Arc.add A't3p5
    (Arc.add A't4p7 Arc.empty))).

Definition mP'1:=marking P'1 1.
...
Definition mP'7:=marking P'7 0.
Definition InitMarkR := ...
```

We can now express the correctness lemma in Coq that corresponds to lemma 2.7. We do not give here the sequence of tactics that constitutes the proof. The proof uses techniques stated in section 3 and corresponds to the mathematical proof in section 2.4.

```
Lemma subnet_refined: Included places P P' /\  Included transitions T T'
  /\ Included  (prodT places nat) InitMarkI InitMarkR /\
                    Arc.Subset Apt A'pt /\
  (forall p t n: nat, PT.In p P -> PT.In t T ->
    ~Arc.In (p,t,n) (Arc.diff Apt A'pt))
                  /\ Arc.Subset Atp A'tp /\
  (forall p t n: nat, PT.In p P -> PT.In t T' ->
~Arc.In (t,p,n) (Arc.diff Atp A'tp)).
```

Similarly to the previous formalisation (with Set), we can implement automation tactics. These two approaches are similar, we will discuss the differences in section 5.

4.4 Using the `List` library

As described in section 3.3, we import the required libraries:

```
Require Export List.
```

4.4.1 Formalisation of a place/transition net

As previously, a net is defined in Coq similarly except that places and transitions are defined with a record type. We denote respectively by `P`, `T`, `Apt`, `Atp` and `InitMarkI` the sets of places, transitions, arcs and the initial marking.

```
Record Place : Type := mkPlace
        { Pl : nat}.

Record Transition : Type := mkTransition
        { Tr : nat}.

Definition P1 := mkPlace 1.
...
Definition T1 := mkTransition 1.
...
Definition Ap1t1 := (P1,T1,1).
...
Definition mP1:=(P1,1).
...
Definition P:= P1::P2::P3::P4::P5::nil.
Definition T:= T1::T2::T3::nil.
Definition Apt := Ap1t1::Ap2t2::Ap4t2::Ap4t3::nil.
Definition Atp := At1p2::At2p3::At3p5::nil.
Definition InitMarkI := mP1::mP2::mP3::mP4::mP5::nil.
```

4.4.2 Subnet refinement

The refined net is defined in Coq similarly. We denote respectively by `P'`, `T'`, `Apt'`, `Atp'` and `InitMarkR` the sets of places, transitions, arcs and the initial marking.

```
Definition P'1 := mkPlace 1.
...
Definition T'1 := mkTransition 1.
...
Definition A'p1t1 := (P'1,T'1,1).
...
Definition mP'1:=(P'1,1).
Definition P' := P'1::P'2::P'3::P'4::P'5::P'6::P'7::nil.
Definition T' :=  T'1::T'2::T'3::T'4::nil.
Definition A'pt := A'p1t1::A'p2t2::A'p4t2::A'p4t3::A'p6t3::A'p6t4::nil.
Definition A'tp := A't1p2::A't2p3::A't3p5::A't4p7::nil.
Definition InitMarkR := mP'1::mP'2::mP'3::mP'4::mP'5::mP'6::mP'7::nil.
```

As previously, we can express the correctness lemma in Coq that corresponds to lemma 2.7.

```
Lemma subnet_refined: incl P P' /\ incl T T' /\
    incl InitMarkI InitMarkR /\ incl Apt A'pt /\
  (forall p:Place, forall t:Transition, forall n: nat, In p P ->
     In t T -> ~((In (p,t,n) Apt)/\(~In (p,t,n) A'pt))
                  /\ incl Atp A'tp /\
  (forall p:Place, forall t:Transition, forall n: nat, In p P ->
     In t T' -> ((In (t,p,n) Atp)/\(~In (t,p,n) A'tp))).
```

4.5 Adding colours

All these experimentations were motivated by specifying Coloured Petri Nets (CPN) in order to prove the type refinement stated in section 2.2. We now extend the previous development so as to add colours. Note that proofs are also adapted. Let us illustrate this on the CPN example in figure 3.

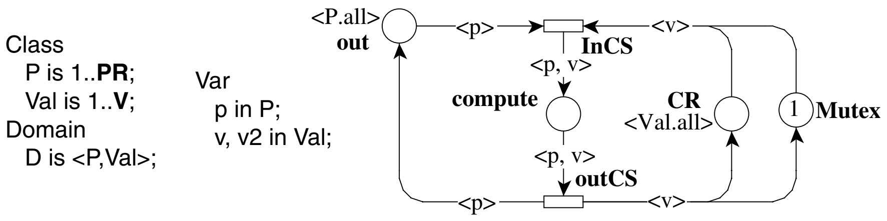

Fig. 3. A Coloured Petri Net example

To deal with these additional data, the question is how to reuse previous specifications adding colours, in other words, which one of these three implementations is best suited? From our experience, the last formalisation, using just simple lists, is the most convincing.

The example of figure 3 can be formalised as follows:

We add to Place and Transition records a field for colours and guards. We define an enumerated type to create a new list with propositions and so on (as a tuple)...:

```
Definition nat_interval (n min max:nat) := min <= n /\ n <= max.

Inductive basic_types : Type :=  cal: Set -> basic_types
                                | propo: Type -> basic_types.

Definition Tuple := list basic_types.

(* arcs values are data_types*)
Inductive data_types : Type :=
          prop : Prop -> data_types
        | tuples : Tuple -> data_types.

(* type of places *)
Record Place : Type := mkPlace
        { P : nat;
          Colors : data_types }.

(* type of transitions *)
Record Transition : Type := mkTransition
        { T : nat;
          Garde : Prop }.
```

Now, we are able to specify places, transitions and arcs:

```
Parameter PR: nat.
Parameter V: nat.
Parameter token: Prop.
Parameter p: nat.
Parameter v: nat.
Definition interval_p p := (nat_interval p 1 PR).
Definition interval_v v := (nat_interval v 1 V).

(* places *)
(* out *)
Definition Piout := mkPlace 1 (tuples ((cal nat)::nil)).
...
Definition listI_pl:= PiMutex::PiCR::Picompute::Piout::nil.

(* transitions *)
(* InCS *)
Definition TiInCS := mkTransition 1 True.
...
```

```
Definition listI_tran:= TioutCS::TiInCS::nil.

(* arcs *)
Definition AiPoutTInCS := (Piout,TiInCS,(interval_p p)).
Definition AiPcomputeToutCS :=
  (Picompute,TioutCS,(interval_p p)/\(interval_v v)).
...
Definition listI_arcs_PT :=
   AiPMutexTInCS::AiPCRTInCS::AiPcomputeToutCS::AiPoutTInCS::nil.
Definition listI_arcs_TP :=
   AiToutCSPMutex::AiToutCSPCR::AiToutCSPout::AiTInCSPcompute::nil.
```

As shown in our previous experiments,we have easily and successfully extended the formalisation of places/transitions net to a simple CPN.

5 Discussion, conclusion and future work

In this section, we will try to explain why we had to experiment with several formalisations. It is quite common when doing formal proofs to choose a formalisation that turns out to be inadequate. In our case, to experiment several formalisations was necessary, at least, for the following reasons:

- our refinement proofs should be generalisable to any Petri net (not only usable for a specific one);
- our refinement proofs should be automatable (abstracting proofs so that we can develop an automation tactic);
- the formalisation should be easily extended to take colours into account;
- the formalisation should easily be modified (and proofs as well which is a more difficult problem), and, in particular, it should be easily adapted to the exchange format of CPN which is in a normalising process [3] and evolving consequently;
- it should be possible to generate the specification with an external tool.

For these reasons, we chose to explore several methods. The "good" choice of formalisation may be compared to a "good" choice of data structures to write a program, with additional difficulties as to elaborate the proof, λ-term "structure" (by Curry-Howard isomorphism), automation,...

The first formalisation (with `Sets`) seems the most natural, because, in a CPN, colours are naturally and often represented by sets. The minor point of this method is that, with the `Sets` library, using finite sets is not natural and requires specific attention. More precisely, extending the formalisation becomes complicated. To deal with finite sets, the `FSets` library is more appropriate, and it was recently developed and integrated in the Coq standard library.

The benefit of the second formalisation (with `FSets`), with respect to the previous one with `Sets`, is that the finite sets library contains many tools (tactics, especially if implementation by red-black trees or AVLs is used). The minor point of this method is that the use of modules and functors yields a difficult automation of specification from an external tool. To add colours "manually" is also not so easy.

The third formalisation (with `List`) is quite simple, it represents sets as lists which is sufficient and allows us to extend it with colours just like adding a field in records. The `List` library also contains some automations that yield easier proofs.

Now, we are able to choose the formalisation that is the most adapted to our goal. This general methodology which consists in making some experimentation before implementing the full problem can be seen as an "investment", rewarded by discovering "the good" approach.

Subnet refinement was formalised and proved in Coq. This first step of formalisation and proof assessed the feasibility and validity of our approach. Automation tactics were also developed.

The common pre-requisite to the three kinds of refinements is the formalisation of a given Petri net. This formalisation is probably the most tedious part of our work and requires a significant automation. The possibility to easily integrate automation at a later stage was a key issue in the work presented in this paper. For example, as seen in Section 4, to define all the places, all the transitions and all the arcs manually is certainly not efficient, especially if the net has more than 50 places/transitions.

We plan to solve this problem using an interface to PNML (Petri Net Markup Language, [3]). PNML is currently being standardised within ISO/IEC 15909-2. It aims at becoming the common language for Petri nets tools, e.g. CPN-AMI [2], CPN-Tools [6] or other tools which handle Petri nets, like FAST [4]. Such files can be directly translated in Coq (as in the third formalisation) so as to generate the places, transitions and arcs.

Our goal is to provide a Petri net refinement specification that enables a user, not familiar with formal proof assistants, to make a formal proof as well. While refinement was extensively studied for other specification languages (e.g., B, VDM/Raise, ...), to our knowledge, there exists neither such a work nor such formalisations of CPN and proofs of refinements.

References

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] CPN-AMI*: Home Page*. http://www-src.lip6.fr/logiciels/mars/CPNAMI/.

[3] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, technology and tools. In *Proc. 24th Int. Conf. Application and Theory of Petri Nets (ICATPN'2003), Eindhoven, The Netherlands, June 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003.

[4] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast acceleration of symbolic transition systems. In *Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003.

[5] *The Coq proof assistant*. http://coq.inria.fr.

[6] *cpntools*. http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

[7] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000.

[8] K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 1: basic concepts*. Monographs in Theoretical Computer Science. Springer, 1992.

[9] G. Lewis. *Incremental specification and analysis in the context of coloured Petri nets*. PhD thesis, University of Hobart, Tasmania, 2002.

[10] C. Lakos and G. Lewis. Incremental state space construction of coloured Petri nets. In *Proc. 22nd Int. Conf. Application and Theory of Petri Nets (ICATPN'01), Newcastle, UK, June 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2001.

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[12] Sam Owre, Natarajan Shankar, and John Rushby. PVS: A Prototype Verification System. In *Proceedings of CADE 11, Saratoga Springs, New York*, June 1992.

[13] Ch. Paulin-Mohring. Inductive definitions in the system Coq, rules and properties. In *Typed Lambda Calculi and Applications (TLCA)*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.