Noname manuscript No.
(will be inserted by the editor)

Coloured Petri net refinement specification and correctness proof with Coq

Christine Choppy · Micaela Mayero · Laure Petrucci

Received: date / Accepted: date

Abstract In this work, we address the issue of the formal proof (using the proof assistant Coq) of refinement correctness for symmetric nets, a subclass of coloured Petri nets. We provide a formalisation of the net models, and of their type refinement in Coq. Then the Coq proof assistant is used to prove the refinement correctness lemma. An example adapted from a protocol example illustrates our work.

Keywords Refinement · coloured Petri nets · theorem proving

1 Introduction

Modelling and analysing large and complex systems requires elaborate techniques and support. To harness the problems inherent to designing and model-checking a large system (such as the state space explosion problem), a specification is often developed step-by-step. First, an abstract model is designed, and its properties are verified. Once it is proven correct, a refinement step takes place, introducing further detail. Such 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 using refinement and hence to start with a more abstract model. It gives a better and more structured view of the system under study. The components within the system are clearly identified and the modelling 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 [9] use the state space of the abstract model to compute the state space of the refined one:

LIPN, UMR CNRS 7030, Université Paris 13, FRANCE
E-mail: firstname.lastname@lipn.univ-paris13.fr

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 symmetric nets, a subclass of coloured Petri nets [7]. Lakos and Lewis [8,9] consider three kinds of Petri nets refinements for coloured Petri nets: node (place or transition) refinement, subnet refinement, and type refinement. Our work provides a formalisation in Coq [5] of both the abstract Petri net and the refined net, as well as refinement correctness lemmas, together with the refinement correctness proof as shown in Figure 1.

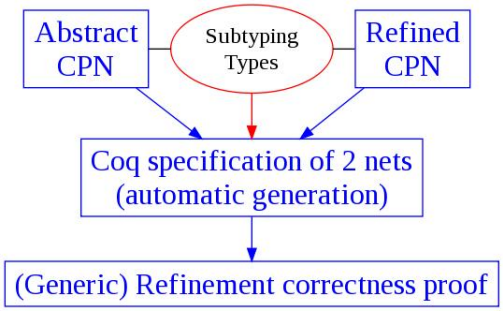

Fig. 1 Refinement model and proof

In a previous work [4], we considered mainly place/transition Petri nets (sketching how coloured nets might be taken into account), and the subnet refinement (the node refinement processing being similar).

In this paper, we address coloured nets, which are more complex in nature, and the formalisation in Coq had to be significantly changed to take the typing issues into account. The *type refinement* formalisation and correctness lemma are also addressed, thus pursuing the initial work of [4]. A protocol example adapted from [7] illustrates our work.

This type refinement is interesting since it allows for the specification of concrete and useful properties in practice. It requires a more complete formalisation, since colour sets (seen as types) are necessary. When compared to place/transition nets, the use of colours decreases the size of nets, leading to more amenable models.

The paper is structured as follows. Section 2 recalls the definitions of coloured Petri nets. Section 3 recalls the different Petri net refinements and provides a correctness lemma for the type refinement. Then, section 4 describes the case study (a protocol example) and formalisations for proving the type refinement of the net in this example. When the proof fails, it can still be used to obtain insight in the model, as explained in section 5. Conclusions and future work are finally discussed in section 6.

2 Coloured Petri nets definition

The definition of coloured Petri nets [8,9] used in this paper is the following:

Definition 1 (Coloured Petri net) A Coloured Petri net $\mathcal{N}$ is an 8-tuple $\mathcal{N} = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ such that:

1. P is a set of places
2. T is a set of transitions, such that $P \cap T = \emptyset$
3. A is a set of arcs, such that $A \subseteq (P \times T) \cup (T \times P)$
4. C: $P \cup T \rightarrow \Sigma$ where Σ is a universe of non-empty colour sets (or types), determines the colours of places and the transition modes.
5. E: $A \rightarrow \Phi\Sigma$ yields the arc inscriptions, such that $E(p,t), E(t,p)$: $C(t) \rightarrow \mu C(p)$
6. $\mathbb{M} = \mu\{(p,c) | p \in P, c \in C(p)\}$ is a set of markings that associate a multi-set of values c with each place p of P.
7. $\mathbb{Y} = \mu\{(t,c) | t \in T, c \in C(t)\}$ is a set of steps (multisets of transitions with their firing mode).
8. M_0 is the initial marking, $M_0 \in \mathbb{M}$.

where $\Phi\Sigma$ is a function over Σ defined by $\Phi\Sigma = \{X \rightarrow Y \mid X, Y \in \Sigma\}$ and $\mu X = \{X \rightarrow \mathbb{N}\}$ are multisets over a set X, and $\mathbb{N}$ is the set of natural numbers.

In the example in Figure 2, the marking of place `PacketsToSend` is the multiset `1‘1++1‘2++1‘3++1‘4++1‘5++1‘6`, where `1‘6` denotes one occurrence of value 6, and `++` denotes the multiset addition operator.

Definition 2 [8] The incremental effects $E^+, E^- : \mathbb{Y} \rightarrow \mathbb{M}$ of the occurrence of a step Y are given by:

1. $E^-(Y) = \sum_{(t,m) \in Y} \sum_{(p,t) \in A} \{p\} \times E(p,t)(m)$
2. $E^+(Y) = \sum_{(t,m) \in Y} \sum_{(t,p) \in A} \{p\} \times E(t,p)(m)$

E^- defines the input arc inscriptions while E^+ defines the output arc inscriptions.

Type refinement modifies the information carried by the tokens (a colour is a value of a token) while the net structure is unchanged. Type refinement brings additional information, which may be done e.g. by adding components in a tuple, or by representing an abstract data type by a more concrete one. The properties of the refined type should be preserved, that is if type A is refined by type B, then type B should satisfy the properties of A after an adequate syntactic translation. As for nets, it should always be possible to associate a behaviour of the abstract net with a behaviour of the refined one. The type refinement issue is associated with the issue of abstraction and implementation in the context of formal specifications (e.g. algebraic specifications [12]), and with studies on subtyping in the context of object-oriented programming languages [11, 3]. In this work (as in the work of Lakos [8,9]), the type refinement considered adds components in a tuple.

Since coloured Petri nets can use very general types and functions over these types which are thus not amenable, we here restrict ourselves to the *symmetric Petri nets* subclass [**?**]. Symmetric nets are defined as coloured Petri nets that allow only the use of particular types and functions: enumerated types, booleans, integer intervals, tuples and combinations of these, as well as the associated functions. We actually also handle lists of such types that can easily be manipulated by the Coq theorem prover.

3 Definitions of refinements

As mentioned previously, Lakos and Lewis [8,9] consider three kinds of Petri nets refinements, node (place or transition) refinement, subnet refinement, and type refinement. *Node refinement* consists in replacing a place (transition) by a place- (transition-) bordered subnet. *Subnet refinement* consists in adding net components (places, transitions and arcs or even additional tokens). In this section the definition of type refinement is recalled, and we give the corresponding correctness lemma. The lemmas for subnet and node refinements can be found in [4]. Up to now, very little work has been achieved concerning type refinement.

In the following definition of *type refinement* [8], $\mathcal{N}_a = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ is the abstract net and $\mathcal{N}_r$ is the refined net.

Definition 3 (Type refinement) A morphism $\phi : \mathcal{N}_a \rightarrow \mathcal{N}_r$ is a *type refinement* if:

1. ϕ is the identity function on P, T, A, i.e. $\forall p \in P$: $\phi(p) = p$, etc.
2. $\forall x \in P \cup T$: $C(x) <: \phi(C)(x)$, i.e. $C(x)$ is a subtype of $\phi(C)(x)$
3. $\forall x \in P \cup T$: $\forall c \in C(x)$: $\phi(1`(x,c)) = 1`(x, \Pi_{\phi(C)(x)}(c))$, where Π is a projection function
4. $\forall (p,t) \in A$: $\forall (t,c) \in \mathbb{Y}$:
   $\phi(E^-(1`(t,c)))(p) = \Pi_{\phi(C)(p)}(E(p,t)(c)) = \phi(E)(p,t)(\Pi_{\phi(C)(t)}(c))$
   $\forall (t,p) \in A$: $\forall (t,c) \in \mathbb{Y}$:
   $\phi(E^+(1`(t,c)))(p) = \Pi_{\phi(C)(p)}(E(t,p)(c)) = \phi(E)(t,p)(\Pi_{\phi(C)(t)}(c))$

The following interpretation will be used in section 4.2 to formalise the type refinement in Coq:

Lemma 1

1. *The network structure (places, transitions and arcs) is kept unchanged, i.e.* $P = P'$, $T = T'$, $uA = uA'$ *where* P', T' *and* uA' *are resp. the sets of places, transitions and arcs (without their associated type) of the refined net while* P, T *and* uA *are those of the abstract net.*
2. *For any token* $1`(x', c')$, *of place* $x'$ *for colour* $c'$ *in the initial marking of the refined net, there exists a corresponding token* $1`(x, c)$ *in the initial marking of the abstract net. They must be such that both the subtyping and projection relations (resp. denoted* $<:$ *and* $\prod$*) are satisfied:* $c <: c'$ *and* $c = \prod_c(c')$.
3. *The arc expressions are refined according to the token refinement:* $\prod_{C_r(p)}(C_a(arc)) = C_r(arc)(C_r(t))$.

According to the formal definition of type refinement in [10], the net structure is unchanged. Since type refinement consists in incorporating additional information in token values, a token type in the refined net is a subtype of the one in the abstract net.

4 Case study: the simple protocol

4.1 Description

In this section, the correctness lemma is illustrated by a simple protocol example adapted from [7].

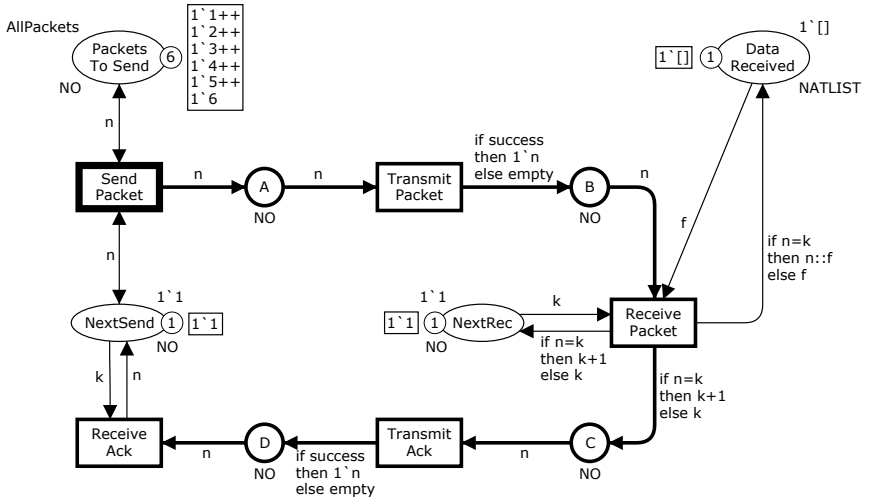

Fig. 2 Example of a simple protocol

```
colset BOOL = bool;
colset NO = int;
colset NATLIST = list NO;
var n,k : NO;
var f : NATLIST;
var success : BOOL;
val AllPackets = 1‘1++1‘2++1‘3++1‘4++1‘5++1‘6;
```

Fig. 3 Colour set declarations for the simple protocol

Figure 2 describes this simple protocol with the associated type (also called colour sets) declarations in figure 3. The left-hand side part models the sender, the right-hand side the receiver, while the middle part represents the network. The sender state is modelled by two places: *PacketsToSend* and *NextSend*. The receiver state is modelled by the *DataReceived* place. Places A, B, C and D constitute the network.

Note that place *PacketsToSend* is initially marked by six tokens with integer values. The textual inscription under a place is called "the colour set" of this place, which represents the available set of token colours. For example, the tokens in place *NextSend* always have an integer value. Here, the colour set `NO` is used to model sequence numbers. The inscription at the top right of place *NextSend* specifies that the initial marking of this place contains a single token with colour (value) 1, while the rectangle on the right exhibits the current marking and the circle contains the number of tokens. Informally, 1‘1 means that the data packet number 1 is to be sent. Finally, we will eventually obtain in place *DataReceived* a list of natural numbers: $[6, 5, 4, 3, 2, 1]$. Let us note that arc expressions yield token values together with their multiplicity. However, when the multiplicity is 1, it is omitted, thus n denotes 1‘n.

This example is refined by associating additional information with tokens (while the net structure in terms of places and transitions is unchanged). The refined net is presented in Figure 4 and the associated colour sets in Figure 5.

The colour sets of places *PacketsToSend*, A, B and *DataReceived* are extended from `NO` to `NOxDATA` which is defined as the cartesian product of the sets describing

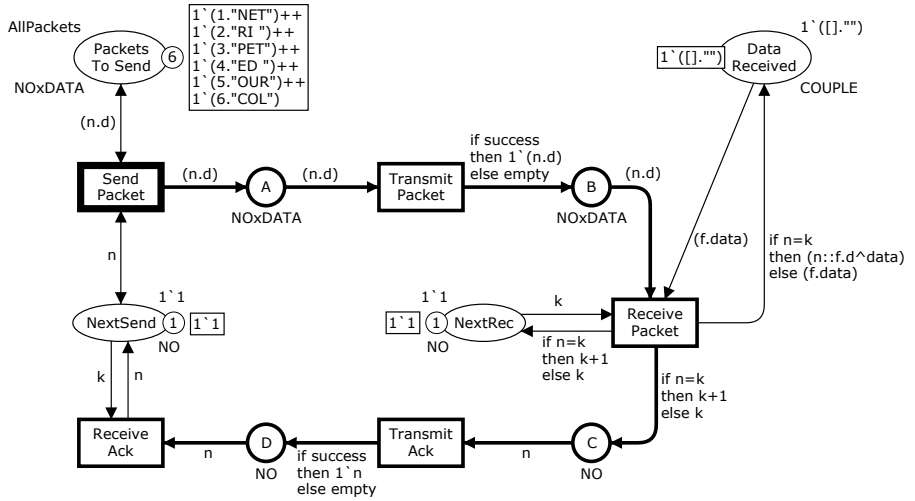

Fig. 4 Refined protocol example

```
colset BOOL = bool;
colset NO = int;
colset NATLIST = list NO;
colset DATA = string;
colset NOxDATA = product NO * DATA;
colset COUPLE = product NATLIST * DATA;
var n,k : NO;
var d,data : DATA;
var f : NATLIST;
var success : BOOL;
val AllPackets = 1‘(1,"NET")++1‘(2,"RI ")++1‘(3,"PET")
               ++1‘(4,"ED ")++1‘(5,"OUR")++1‘(6,"COL");
```

Fig. 5 Colour set declarations for the refined protocol

types `NO` and `DATA`. Note that here some places are not refined, e.g. *NextSend* is still of type `NO`.

The type refinement achieved in Figure 4 not only changes the type of tokens and places but also modifies the arcs expressions accordingly. Note that there exists a subtyping relation between e.g. n and (n, d) (the colour sets are given in Figures 3 and 5) on the arc between *PacketsToSend* and *SendPacket*.

4.2 Formalisation and proof in Coq

The simple protocol example is now formalised. Tokens carry complex information, and several functions are required to certify the system. In this example, the arc expressions are of four possible types (all with an integer multiplicity).

nat $*$	(nat)
	(nat $*$ string)
	(list nat)
	(list nat $*$ string)

In Coq, these kinds of arcs are defined by an inductive type:

```
Definition nat_Tuple := list nat.
Definition string_Tuple := list string.

Inductive arc_type : Type :=
  |bi_types: nat*nat -> arc_type
  |tri_types: nat*(nat*string) -> arc_type
  |bi_n_tuples: nat*nat_Tuple -> arc_type
  |tri_tuples: nat*(nat_Tuple*string_Tuples) -> arc_type.
```

Then, places and transitions are indexed by natural numbers:

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

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

We now present an excerpt of the definitions of places, transitions and arcs for our example. Sets of places, transitions or arcs (both the untyped arc, i.e. the edge in the graph, and the arc expression) are represented by lists:

```
Definition P1_PacketsToSend := mkPlace 1.
...
Definition list_P:= P1_PacketsToSend::P2_A::P3_B::P4_NextRec::
                    P5_DataReceived::P6_C::P7_D::P8_NextSend::nil.

Definition T1_SendPacket := mkTransition 1.
...
Definition list_T:= T1_SendPacket::T2_TransmitPacket::T3_ReceivePacket::
                    T4_TransmitAck::T5_ReceiveAck::nil.

Definition uAP1T1 := (P1_PacketsToSend,T1_SendPacket).
...
Definition AP1T1 := (P1_PacketsToSend,T1_SendPacket,bi_types (1,n)).
...
Definition list_APT := AP1T1::AP2T2::AP3T3::AP4T3::AP5T3::
                       AP6T4::AP7T5::AP8T5::AP8T1::nil.
Definition list_ATP := AT1P1::AT1P2::AT1P8::AT2P3::AT3P4::
                       AT3P5::AT3P6::AT4P7::AT5P8::nil.
Definition list_uATP := uAT1P1...
...
Definition list_APT' := A'P1T1::A'P2T2::A'P3T3::A'P4T3::A'P5T3::
                        A'P6T4::A'P7T5::A'P8T5::A'P8T1::nil.
Definition list_ATP' := A'T1P1::A'T1P2::A'T1P8::A'T2P3::A'T3P4::
                        A'T3P5::A'T3P6::A'T4P7::A'T5P8::nil.
```

The most interesting aspect of type refinement is due to arc expressions. Type refinement can be seen as a relation between types, which is subtyping. For example, the following table presents subtyping relations involved in our refinement of the simple protocol (note that the first line is unchanged by the refinement, and this still needs to be checked).

ARC EXPRESSIONS	EXAMPLE OF ARC VALUES	VALUE TYPE	Coq `arc_type`
if n=k then k+1 else k	1‘6	nat*nat	`bi_types`
n (n, d)	1‘6 1‘(6, ”COL”)	nat*nat nat*(nat*string)	`bi_types` `tri_types`
if n=k then n::f else f if n=k then (n::f, d∧data) else (f, data)	1‘[6] 1‘([6], ”COL”)	nat*list nat nat*(list nat*list string)	`bi_n_tuples` `tri_tuples`
f (f, data)	1‘[6]) 1‘(6, ”COL”)	nat*list nat nat*(list nat*list string)	`bi_n_tuples` `tri_tuples`

The subtyping relation must be formalised for this example. We begin by defining a function `is_sub` which gives a relation between types of `arc_type`. This relation is then extended to tuples (`is_sub_tupl_apt`) and lists of tuples (`is_sub_l_apt`) for describing arcs from places to transitions. Similar extensions are defined for arcs from transitions to places.

```
Definition is_sub (subtyp:arc_type)(typ:arc_type) : Prop :=
  match subtyp, typ with
  | (bi_types _), (bi_types _) => True
  | (tri_types _), (bi_types _) => True
  | (tri_tuples _), (bi_n_tuples _) => True
  | _, _ => False
  end.

Definition is_sub_tupl_apt (subtupl: Place * Transition * arc_type)
           (tupl: Place * Transition * arc_type) : Prop :=
           (is_sub (snd subtupl) (snd tupl)).

Fixpoint is_sub_l_apt (subl: list (Place * Transition * arc_type))
         (l: list (Place * Transition * arc_type)) {struct subl} : Prop :=
    match subl, l with
     | nil, nil => True
     | (cons a tla), (cons b tlb) =>
             (is_sub_tupl_apt a b) /\ (is_sub_l_apt tla tlb)
     | _, _ => False
    end.
```

In this code, several operators are used: `snd` returns the second component of a pair, `nil` denotes the empty list and `cons` is the list constructor, adding an element at the head of a list.

The type refinement correctness lemma can now be written, with the help of lemma 1 (where, for the sake of readability, we indicate in parenthesis to which item the Coq code relates):

```
Lemma type_colour_refined:
      eqlist Place list_P list_P' /\                        (1.)
      eqlist Transition list_T list_T' /\                   (1.)
      eqlist (Place*Transition) list_uAPT list_uAPT' /\     (1.)
      eqlist (Transition*Place) list_uATP list_uATP' /\     (1.)
      eqlist (list (nat*nat)) list_MP (hd_list list_MP')/\ (2.)
```

```
      is_sub_l_apt list_APT' list_APT /\                    (3.)
      is_sub_l_atp list_ATP' list_ATP.                      (3.)
```

where `eqlist` is an equality between lists and $l = l'$ is equivalent to $l \subseteq l'$ and $l' \subseteq l$, `list_MP` and `list_MP'` define the initial markings, and function `hd_list` is defined as follows:

```
Fixpoint hd_list_couple (l:list (nat*(nat*string))):=
     match l with
       | nil=>nil
       |(a,(b,c))::tl=>(a,b)::(hd_list_couple tl)
     end.

Fixpoint hd_list (l:list (list (nat*(nat*string)))):=
      match l with
        |nil=>nil
        |a::tl=>(hd_list_couple a)::(hd_list tl)
      end.
```

Thanks to our simple and general formalisation, the formal correctness proof is almost automatic.

```
Proof.
repeat split;unfold incl;tauto.
Qed.
```

Note that this simple formalisation was obtained after carefully studying different possibilities for encoding Petri net elements in Coq which are detailed in [4]. Moreover, the proof could be simplified using powerful constructs such as the `split` tactic, which is particularly well-suited for our purposes. This tactic applies to inductive types with a single constructor, which is the case for the `/\` operator in the lemma `type_colour_refined`. While in [4] we used more traditional and complex proofs, we since got a better insight that lead us to produce a much simpler and more powerful Coq proof that is independent of the example considered.

The full development is available at `http://www-lipn.univ-paris13.fr/~mayero/CPNCoq/Jensen_protocol_JNASA.v`.

5 When the proof fails

When the proof fails, it still gives valuable information as regards the refinement to be proven: either the lists representing the net graph elements (places, transitions, or arcs) do not match, and the refinement relation does not hold ; or the error occurs when examining arc expressions. It may then be the case that the refinement property does not hold, but also that the type refinement between the supposedly refined and abstract arc expression cannot be automatically proven.

In order to try to know in which case we are, we have to render the Coq proof less automatic. For example, suppose that the subtyping function `is_sub` is erroneous. In

this function, let us now assume that the following line is missing:
`| (tri_tuples _), (bi_n_tuples _) => True`
This means that: `| (tri_tuples _), (bi_n_tuples _) => False`

In this case, the proof script "repeat split;unfold incl;tauto." provides the following error message:

```
"File "./Jensen_protocol_NFM_detect_error.v", line 266, characters 25-30
User error: Tauto failed"
```

And the goal remains unchanged:

```
type_colour_refined < Show.
1 subgoal

  ============================
   eqlist Place list_P list_P' /\
   eqlist Transition list_T list_T' /\
   eqlist (Place * Transition) list_uAPT list_uAPT' /\
   eqlist (Transition * Place) list_uATP list_uATP' /\
   eqlist (list (nat * nat)) list_MP (hd_list list_MP') /\
   is_sub_l_apt list_APT' list_APT /\ is_sub_l_atp list_ATP' list_ATP
```

From this error message, we know that the problem is detected by the tactic `tauto`. Due to the high level of proof automation, we have to begin by finding which goal is concerned. To do so, it is possible in Coq to catch errors using `try` :

```
repeat split;unfold incl;try tauto.
```

If the proof does not terminate then Coq leaves subgoals to be proved (instead of failing). The user can interactively see where the problems are. In our example, it is the last two subgoals :

```
type_colour_refined < repeat split;unfold incl;try tauto.
2 subgoals

  ============================
   is_sub_tupl_apt A'P5T3 AP5T3

subgoal 2 is:
 is_sub_tupl_atp A'T3P5 AT3P5
```

We can further detail the proof:

```
unfold is_sub_tupl_apt;unfold is_sub;simpl.
```

Thanks to the interactive interface of Coq (called toplevel), we can see that we have to prove False (instead of True)! This is due to the fact that the function `is_sub` is not correct.

```
type_colour_refined < unfold is_sub_tupl_apt;unfold is_sub;simpl.
2 subgoals

  ============================
   False
```

This means that a subtyping property is missing.

6 Conclusion

When modelling and validating critical systems, one often proceeds in a step-by-step fashion: a first abstract model is designed and validated ; it is then refined so as to take into account additional details ; and this process is repeated as many times as necessary. In order to guarantee that the behaviour of the system is preserved by refinement, it should obey some rules. Three kinds of refinements of coloured Petri nets were formally defined in [9]. Our aim here was to show that the proof of refinement — i.e. that a refined net actually is a refinement of an abstract net — can be automated using theorem-proving techniques, thus avoiding error-prone and lengthy manual proofs.

Previous work focussed on two kinds of refinements: node refinement and subnet refinement, while the third one was scarcely mentioned. This paper has shown that when restricting coloured Petri nets to an appropriate subclass, type refinement can also be handled.

This work confirms that our choices of formalisation (with a first version in [4]) are suitable. The prerequisite to the 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. Since the refinement issues we tackle are meant to be integrated within a step-by-step modelling process, the refined net should be designed by the user starting from the abstract net. Therefore, places and transitions that are in both nets should remain exactly the same and can be identified by their name. Hence, we do not address the problem of proving that a net is a refinement of another one, starting from arbitrary nets.

The possibility to easily integrate automation is 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. To solve this problem, an interface to PNML (Petri Net Markup Language, [2]) is under development, using PNML Framework [**?**]. PNML is part of the ISO/IEC 15909-2 standard (since November 2009). It aims at becoming the common language for Petri nets tools, e.g. CPN-AMI [1], CPN-Tools [6] or other tools supporting Petri nets. Such files can be directly translated into Coq to generate the places, transitions and arcs.

We think that our method scales up rather well. Indeed, the proof is generic and does not change with the nets considered. The only modifications are subtyping re-

lations and type definitions. Moreover, when proceeding step-by-step, refinements are applied one at a time. Therefore, the nets to be considered are only slightly different.

To complete this work, we should consider refinement as part of a modular design process. In such a framework, a type refinement can affect several modules which could be checked separately for refinement, and one must ensure that type refinement has been applied consistently in all modules.

Acknowledgements The implementation of this work in Coq was achieved with the help of Yibei Yu, a trainee supervised by the authors. We would like to thank Charles Lakos for fruitful discussions.

References

1. cpn-ami*: Home Page*. `http://www-src.lip6.fr/logiciels/mars/CPNAMI/`.
2. 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.
3. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.
4. Christine Choppy, Micaela Mayero, and Laure Petrucci. Experimenting Formal Proofs of Petri Nets Refinements. *Electr. Notes Theor. Comput. Sci.*, 214:231–254, 2008.
5. *The Coq proof assistant*. `http://coq.inria.fr`.
6. *cpntools*. `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`.
7. Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets, Modelling and Validation of Concurrent Systems*. Monograph to be published by Springer Verlag, 2008.
8. Charles Lakos. Composing abstractions of coloured Petri nets. In Nielsen, M. and Simpson, D., editors, *Lecture Notes in Computer Science: 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), Aarhus, Denmark, June 2000*, volume 1825, pages 323–345. Springer-Verlag, 2000.
9. Charles Lakos and Glen 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.
10. Glen Lewis. *Incremental specification and analysis in the context of coloured Petri nets*. PhD thesis, University of Hobart, Tasmania, 2002.
11. Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 118–141, London, UK, 1993. Springer-Verlag.
12. Fernando Orejas, Marisa Navarro, and Ana Sanchez. Algebraic implementation of abstract data types: a survey of concepts and new compositionality results. *Mathematical Structures in Computer Science*, pages 33–67, 1996.