Coloured Petri Nets Refinements

C. Choppy, L. Petrucci, and A. Sanogo

LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
`{Christine.Choppy, Laure.Petrucci, Alfred.Sanogo}@lipn.univ-paris13.fr`

Abstract. Coloured Petri nets allow for modelling complex concurrent systems, but the specification of such systems remains a challenging task. Two approaches are generally used to lessen these difficulties: decomposition and refinement.
Charles Lakos proposed three kinds of refinements for coloured Petri nets: type refinement, subnet refinement, and node (place or transition) refinement. This paper proposes new rules widening the scope of both type and transition refinements.

1 Introduction

Coloured Petri nets are a specification language which presents the advantages of both a graphical description, giving an easy understanding of the model, and a formal semantics allowing for formal analysis techniques. However, as is the case for many specification languages, the specification of a system remains a difficult task. A way to alleviate these difficulties consists in using refinement techniques.

First, a rather simple model of the system is built, at a high level of abstraction, with few details. This abstract model constitutes a general description of the system. An incremental process of successive refinements is then applied, adding new details in a stepwise manner. The model is thus enhanced until all the expected behaviour and properties are taken into account. At each step, the abstract model is replaced by a refined one.

For defining refinements, the following questions should be addressed:

– which relation should exist between the abstract and the refined models?
– what are the necessary conditions on the coloured Petri net models for this relation to hold?
– which transformations of the abstract net satisfy these refinement conditions?

Since a model is characterised by its observed behaviour, the comparison between abstract and refined model will rely on it. Several notions of equivalence and order have been proposed in the literature. In particular, Lakos and Lewis [6, 8, 5, 7] propose that "*a model R is a refinement of a model A if all behaviour in R has a corresponding behaviour in A*" (thus a refinement, while it may introduce further details, cannot introduce behaviours that would be new to the abstract model; this is useful to be able to explore the state space in a modular and still meaningful way). They express this relation between coloured Petri nets as a system morphism (behavioural morphism) from the refined model to the abstract one. Three kinds of refinements were proposed, and the system morphisms defined accordingly: type refinement, subnet refinement and node (place or transition) refinement. Transformation rules on the net structure, the colours and firing modes, that respect these morphisms, complete their work by providing practical refinement mechanisms.

This paper extends these coloured Petri nets refinements, and more specifically the type and transition refinements. Type refinement includes two constraints: the subtype relation defined by Liskov and Wing [9] as well as Lakos' refinement condition. Four operations on types are now permitted: addition of a component in a tuple, constraining a component value, addition and modification of functions. The conditions for these operations to satisfy the refinement constraints are checked. For node refinement, a new refinement rule satisfying Lakos's constraints is introduced: alternate transitions.

The paper is organised as follows. In Section 2 definitions of coloured Petri nets and system morphisms are recalled. Then, Section 3 recalls the refinements defined by Lakos and Section 4 the subtyping relation of Liskov and Wing. Our new refinement rules are then defined and proven correct in Sections 5 and 6. They are implemented in a tool for Petri net design, described in Section 7.

2 Coloured Petri Nets and Morphisms

In this section, we recall the necessary definitions and notations from [6].

2.1 Coloured Petri Nets

For a type universe Σ, we denote the set of functions from one type of Σ to another by $\Phi\Sigma = \{X \to Y \mid X, Y \in \Sigma\}$ and by $\mu X = \{X \to \mathbb{N}\}$ the set of multisets over a type $X \in \Sigma$.

Definition 1. *A* coloured Petri net *is a tuple* $N = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ *where:*

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 *is a* colour function *which associates a type with each place and transition:* $C : P \cup T \to \Sigma$*;*
5. E *is a function associating an* expression *with each arc:* $E : A \to \Phi\Sigma$ *with* $E(p,t), E(t,p) : C(t) \to \mu C(p)$*;*
6. $\mathbb{M}$ *is the* set of markings*:* $\mathbb{M} = \mu\{(p,c) \mid p \in P, c \in C(p)\}$*;*
7. $\mathbb{Y}$ *is the* set of steps*:* $\mathbb{Y} = \mu\{(t,c) \mid t \in T, c \in C(t)\}$
8. $M_0 \in \mathbb{M}$ *is the* initial marking.

For a node $x \in P \cup T$, we denote by

- ${}^\bullet x$ the *preset* of x, i.e. ${}^\bullet x = \{y \in P \cup T | (y, x) \in A\}$;
- $x^\bullet$ the *postset* of x, i.e. $x^\bullet = \{y \in P \cup T | (x, y) \in A\}$.

In the following, E^- denotes the expressions of arcs from places to transitions and E^+ from transitions to places. The firing rule of a coloured Petri net is now defined.

Definition 2. *Let N be a coloured Petri net. A step $Y \in \mathbb{Y}$ is* firable *from a marking $M \in \mathbb{M}$, denoted $M[Y\rangle$ iff $M \geq E^-(Y)$.*

In order to replace a node of a Petri net by a subnet during the refinement process, the connections between these and their environment must be considered. Therefore, we now define the border nodes on each side, adapted from [6].

Definition 3. *Let N be a Petri net, and N' a subnet of N.*

1. *The* input border of N' *is* $inpbdr(N') = \{x \in P' \cup T' | \exists y \in (P \cup T) \setminus (P' \cup T') : y \in {}^\bullet x\}$*;*
2. *The* output border of N' *is* $outbdr(N') = \{x \in P' \cup T' | \exists y \in (P \cup T) \setminus (P' \cup T') : y \in x^\bullet\}$*;*
3. *The* border of N' *is* $bdr(N') = inpbdr(N') \cup outbdr(N')$*;*
4. *The* set of input environment nodes of N' *is* $inpenv(N') = \{y \in (P \cup T) \setminus (P' \cup T') | \exists x \in P' \cup T' : y \in {}^\bullet x\}$*;*
5. *The* set of output environment nodes of N' *is* $outenv(N') = \{y \in (P \cup T) \setminus (P' \cup T') | \exists x \in P' \cup T' : y \in x^\bullet\}$*;*
6. *The* set of environment nodes of N' *is* $env(N') = inpenv(N') \cup outenv(N')$.

Example: Let us consider the net in Figure 1, with the subnet in the large circle. Its border sets are:

- $inpbdr(N') = \{inp_1, inp_2\}$;
- $outbdr(N') = \{out_1, out_2\}$;
- $bdr(N') = \{inp_1, inp_2, out_1, out_2\}$;
- $inpenv(N') = \{te_1, te_2\}$;
- $outenv(N') = \{ts_1, ts_2\}$;
- $env(N') = \{te_1, te_2, ts_1, ts_2\}$.

2.2 System Morphisms

System morphisms were defined in [6] in order to capture behaviour compatibility between nets. In such a scheme, firing several transitions of N_r can correspond to the firing of a single transition in N_a. A step is then said to be complete if all border transitions of $N_r \setminus N_a$, i.e. the subset that remains after N_a has been removed from N_r, are fired with the same firing mode.

Definition 4. *Let $\phi : N_r \to N_a$. ϕ is a* system (behavioural) morphism *if:*

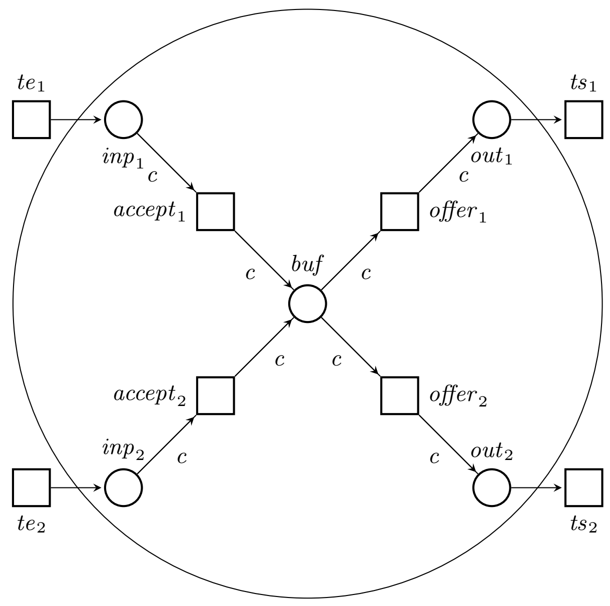


Fig. 1. Example of border nodes

1. *ϕ is surjective on P_a, T_a, A_a;*
2. *ϕ is linear and total on the set of markings $\mathbb{M}_r$ and the set of steps $\mathbb{Y}_r$;*
3. *$\forall M_r \in [M_0\rangle_r$, $\forall Y_r \in \mathbb{Y}_r$: if Y_r is complete, can be decomposed into Y_1, Y_2 ... Y_n, and is realisable from marking M_r, then $\phi(Y_r)$ can be decomposed into $\phi(Y_1), \phi(Y_2)$ ...$\phi(Y_n)$, and is realisable from marking $\phi(M_r)$.*
4. *$\forall M_r \in [M_0\rangle_r, \forall Y_r \in \mathbb{Y}_r : Y_r$ is complete $\Rightarrow \phi(M_r + E_r^+(Y_r) - E_r^-(Y_r)) = \phi(M_r) + \phi(E_r^+)(\phi(Y_r)) - \phi(E_r^-)(\phi(Y_r))$.*

In Definition 4, (3) indicates that when a step in N_r is decomposable, the corresponding step can be obtained by projection on N_a, while (4) states that only the effect of a step can be projected as well.

Definition 5. *Let $\phi : N_r \rightarrow N_a$ be a system morphism. A step Y_r in N_r is said to be* complete *if $\forall t_a \in T_a : \forall t_r \in bdr(T_r \setminus T_a) : Y_r(t_r) = \phi(Y_r)(t_a)$*

3 Existing Coloured Petri Nets Refinements

Three types of coloured Petri nets refinements are defined by Lakos in [6]: type refinement, node refinement, and subnet refinement.

3.1 Type Refinement

Type refinement is used when it is necessary to detail further the description of the information carried by tokens, and the transitions' firing modes. Refinement then consists in replacing an abstract token type by another one, more detailed, called the refined type. The net structure remains unchanged. The type modification addressed by Lakos is the addition of a component.

3.2 Subnet Refinement

Subnet refinement consists in adding new elements (places, transitions and arcs) to the net.

[6] also considers extending type domains for places and transitions in the abstract net, as a subnet refinement.

3.3 Node Refinement

Node refinement is used when the modeller wishes to give additional information about one of the net elements (place or transition) by expliciting it further. It can thus be a place refinement if details concern a place (superplace) or transition refinement if a transition is concerned (supertransition). Lakos defined canonical refinements for both of these cases.

In its canonical form, a place refinement replaces a place by a place-bordered subnet, whereas a transition refinement replaces a transition by a transition-bordered subnet.

4 The Sub-typing Relation

A subtype relation was defined in [9] by Barbara Liskov and Jeannette Wing so that the supertype properties should be preserved by the subtype. The properties considered are invariants (that should be true for any object state) and "historical" properties (true on a state sequence). The substitutability principle states that a subtype object could substitute a supertype object.

Types are denoted by a triple $\langle O, V, M\rangle$ where O is a set of objects, V is a set of values, and M is a set of methods.

The type specification should contain the following information:

- the description of the set of authorised values;
- for each method, the description of (i) its signature (number and types of arguments, result type and exceptions list), (ii) its behaviour in terms of pre- and post-conditions.

B. Liskov and J. M. Wing distinguish three kinds of methods, *constructors* that return a new object of the type, *observers* that return values of other types, and *mutators* that modify object values.

They also identify two kinds of subtyping relations:

- *Extension subtype* where the subtype extends the supertype by introducing new methods and adding new states (or values).

– *Constrained subtype* where the subtype constrains the supertype with more details on the methods or on the supertype values. When the supertype specification keeps open several possibilities, subtyping may reduce or eliminate some of them.

Considering the two kinds of subtyping relations above, we consider here the following operations on types:

– *add a component* to a record
– *fix a component value*
– *add a method*
– *modify a method.*

These operations on types appear in the subtype relations as follows.

1. *Extension Subtype* The operations on types considered are the introduction of new methods and/or the addition of new states.
   – New method introduction
     • if the method introduced is a *constructor*, it should take into account the invariant preservation;
     • if the method introduced is an *observer*, this has no consequence on the fact that the subtype preserves the supertype properties;
     • if the method introduced is a *mutator*, this may have consequences on the fact that the subtype should preserve the supertype properties. After a mutator is invoked, the new object value should belong to the set of authorized type values (according to invariants and historical properties).
   – Addition of new object states (or adding a new variable in records): this has no effect on properties preservation, and the abstraction function forgets the added variable.
2. *Constrained Subtype* The operations considered here are constraints on the object values and/or method modification with the aim to add more precision.
   – Constraints on the object values (this may correspond to fixing a variable value in a record). This may be achieved by defining the set of values for the subtype objects as a subset of the supertype values and this preserves the supertype properties.
   – Method modification. This modification must comply with the subtyping rules defined in [11], that are a rule on the signature (same number of arguments, contravariance of arguments, covariance of result and exception rule), and a rule on methods.

Let us recall that if σ is a subtype of τ and m_τ is the τ method corresponding to method m_σ of σ :

– arguments contravariance states that m_τ and m_σ should have the same number of arguments, and type α_i of the ith argument in m_τ should be a subtype of type β_i of the corresponding argument in m_σ.
– either m_τ and m_σ return a result or do not return a result. If both methods return a result, the result covariance states that the type of the result returned by m_σ should be a subtype of the type of the result returned by m_τ.

5 New Rules for Type Refinement

An initial definition of *type refinement* was proposed by Lakos in [6] requiring that all behaviours of the refined type correspond to an abstract behaviour. This constraint will be referred to as LC in the following (Lakos's constraint). The only operation allowed by Lakos for type refinement is the addition of a component in a tuple. Although this kind of refinement is often used, it is still restrictive in practical cases which allow for more substantial type modifications and introduction of new operators on existing types.

Therefore, in this section, we extend the relation between the refined type and the abstract type by considering the sub-typing relation defined by Liskov and Wing in [10], as well as type modifications.

5.1 Type modifications and behaviour preserving

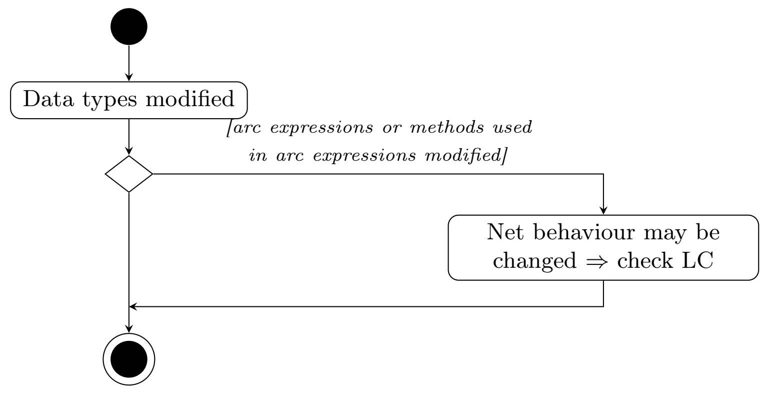

Data types modified
[arc expressions or methods used in arc expressions modified]
Net behaviour may be changed ⇒ check LC

Fig. 2. Summary of arc expressions modifications impact

When performing type refinement, the net structure remains unchanged. Hence the only elements which may change the net behaviour are the values of arc expressions, and transitions guards.

Arc expressions Subsequently to a type refinement, an arc expression function might be modified. Figure 2 summarises the impact of type modification on arc expressions.

In the following, E_r denotes arc expressions in the refined net, and E_a the corresponding arc expression in the abstract net.

Unchanged arc expressions If the arc expression is the same, i.e. $E_r = E_a$, the following cases may occur:

1. adding a component to a tuple, setting the value of a component, adding a method, are changes that respect LC;
2. modification of a method leads to the following situations:
   – E_a does not refer to a method modified by the refinement. The values returned by E_r and E_a are thus the same, and the net behaviour remains unchanged ;
   – E_a refers to methods changed by the refinement. Hence, although the expressions are the same, the values returned by E_r and E_a may differ. Therefore the behaviour can be different as well, and we need to check that the effect of firing for each firing mode in the refined net is the same, once abstracted, as for the corresponding firing in the abstract net.

Modified arc expressions When an arc expression is modified due to the refinement process, values of E_r and E_a may differ, whatever the refinement operation. Hence, adding a component, setting the value of a component, adding a method may all modify the net behaviour. The expression modification should be performed so that it satisfies LC.

Guards associated with transitions Type refinement does not change the expression associated with a guard. However, two cases may occur (see Fig. 3):

– the guard does not refer to a method modified by the type refinement: it has thus no influence on the transition behaviour;
– the guard refers to methods modified by the type refinement. Hence the values returned by the refined and abstract guard expressions may differ. If the guard returns *true* in the refined net, it must also return *true* in the abstract net.

5.2 Example: request for material purchase

The net in Fig. 4(a) models a request for material issued from e.g. a service to the accountant. When material is needed, a request is issued. After the accountants check the request, they order the articles.

In a first approach to modelling the problem, the net describing the overall process can de considered. It uses a neutral colour, as shown in Fig. 4(a).

Then additional detail can be introduced, giving characteristics of the article to order, i.e. its *number*, its designation (*name*), its *origin*, the quantity required (*qty*) and the unitary *price*. The new type is thus obtained by adding components, as shown in the declarations below and Fig. 4(b). In this case, no arc or guard expression is changed, therefore LC is satisfied. It is also consistent with the subtyping conditions.

In order to take into account calls for offers when the price is above a certain level, a method is added which returns the total *amount* of the purchase

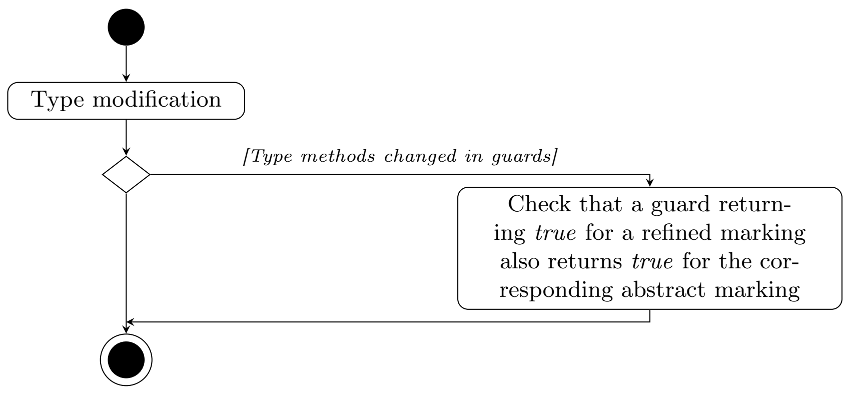
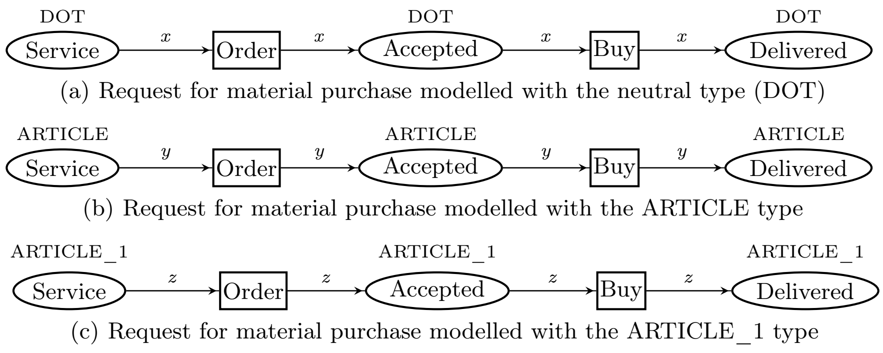

Type modification
[Type methods changed in guards]
Check that a guard returning true for a refined marking also returns true for the corresponding abstract marking

Fig. 3. Summary of the impact of type refinement on guards and net behaviour



(a) Request for material purchase modelled with the neutral type (DOT)

(b) Request for material purchase modelled with the ARTICLE type

(c) Request for material purchase modelled with the ARTICLE_1 type

Fig. 4. Type refinement of a net model of a request for material purchase

$price * qty$ (see type $ARTICLE\_1$) in the declarations and Fig. 4(c)). This modification adds an observer method which returns a value, and is consistent with the subtyping relation. It is not used in arc or guard expressions, and therefore LC holds.

```
Declarations:
DOT is a predefined neutral type
Colset ARTICLE = record number:NAT * origin:STRING *
                 name:STRING * price:Rat * qty:Rat;

Colset  ARTICLE_1 = record number:NAT * origin:STRING *
                    name:STRING * price:Rat * qty:Rat;
fun amount = (#price ARTICLE_1 * #qty ARTICLE_1)::Rat;
var x : DOT;
var y : ARTICLE;
var z : ARTICLE_1;
var name : STRING;
var price : Rat;
var qty : Rat;
```

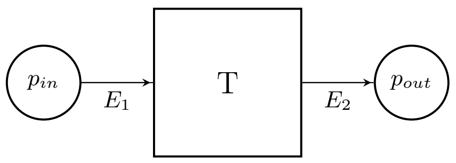
6 New Rules for Transition Refinement: Alternate Transitions

The canonical transition refinement proposed by Lakos, firing an abstract transition features firing a set of sequences of refined transitions, starting with transitions from the input border and ending with transitions from the output border of the refined part.

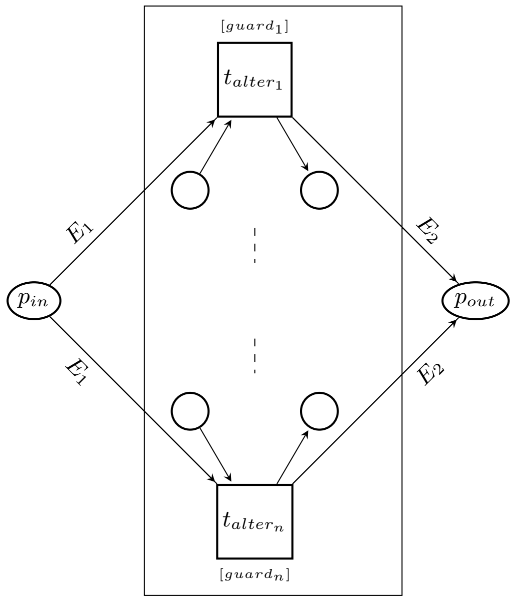
The underlying idea of the new refinement we propose here is to replace an abstract transition by a subnet containing alternative transitions plus internal places and transitions. Each of the alternative transitions is abstracted as the original abstract transition. This refinement aims at splitting a transition describing a general behaviour into several exclusive cases which then handle in more detail specific situations. For example, the net in Fig. 5(b) is a refinement of the one in Fig. 5(a): transition t_a is replaced by a subnet N_{t_a}. To improve readability, t_a has a single input place and a single output place (but this is not a constraint in the general case).

Definition 6. *Let $N_{t_a} = (P_{t_a}, T_{t_a}, A_{t_a}, C_{t_a}, E_{t_a}, \mathbb{M}_{t_a}, \mathbb{Y}_{t_a}, M_{t_{a0}})$ be a subnet. A morphism*
$\phi : N_r = (P_r, T_r, A_r, C_r, E_r, \mathbb{M}_r, \mathbb{Y}_r, M_{r0})$
$\quad \to N_a = (P_a, T_a, A_a, C_a, E_a, \mathbb{M}_a, \mathbb{Y}_a, M_{a0})$
is a refinement with alternative transitions *of $t_a \in T_a$, where N_r is the refined net obtained by replacing abstract transition t_a by N_{t_a}) in the abstract net N_a if:*

1. $\forall p_r \in P_r \setminus P_{t_a} : \forall t_r \in T_r \setminus T_{t_a}$:
   $\phi(p_r) = p_r \wedge \phi(t_r) = t_r \wedge$
   $(p_r, t_r) \in A_r \Rightarrow ((p_r, t_r) \in A_a \wedge E_r(p_r, t_r) = E_a(p_r, t_r)) \wedge$
   $(t_r, p_r) \in A_r \Rightarrow ((t_r, p_r) \in A_a \wedge E_r(t_r, p_r) = E_a(t_r, p_r))$. *Apart from transition t_a the abstract net remains unchanged.*
2. $T_{t_a} = T_{alternative} \cup T_{other}$ *The transitions replacing t_a are of two kinds: the alternatives, and the others.*
3. $\forall t \in T_{alternative},\ {}^\bullet t_a = {}^\bullet t \setminus P_{t_a} \wedge \forall p \in {}^\bullet t_a : E_a(p, t_a) = E_r(p, t)$
   All input places of t_a are also input places of all alternative transitions, and are the only such abstract places.
4. $\forall t \in T_{alternative}, t_a^\bullet = t^\bullet \setminus P_{t_a} \wedge \forall p \in t_a^\bullet : E_a(t_a, p) = E_r(t, p)$
   All output places of t_a are also output places of all alternative transitions, and are the only such abstract places.
5. $\forall M \in \mathbb{M}_r : \forall t' \in T_{alternative}$:
   $\phi(M)[t_a > \ \wedge \ M[t' > \Rightarrow \forall t \in T_{alternative} \setminus \{t'\}, \neg(M[\ t >)$
   There is at most one alternative transition that is firable in a given marking (so that the token flow is preserved). This can be ensured by guards or internal places of the N_{t_a} subnet.



(a) Abstract transition to be refined



(b) Refinement of transition t_a

Fig. 5. Refinement with alternative transitions

6. $\forall t \in T_{alternative} \wedge \forall c \in C_r(t) : \phi(1\,`(t,c)) = 1\,`(t_a, \phi(c))$
   Firing an alternative transition has the same effect as firing t_a. Firing a step in the refined net has the same effect as in the abstract net.
7. $\forall p \in P_r : \forall c \in C_r(p) : \phi(1\,`(p,c)) = 1\,`(p,c)$ *if* $p_r \in P_r \setminus P_{t_a}$, $\phi(1\,`(p,c)) = \emptyset$ *otherwise.*
   The internal marking of N_{t_a} is ignored by the refinement.

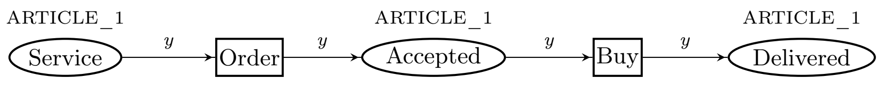
Proposition 1. *A refinement with alternative transitions $\phi : N_r \rightarrow N_a$ is a systems morphism.*

Proof.
According to definition 6(1) ϕ is surjective over P_a, T_a, A_a.

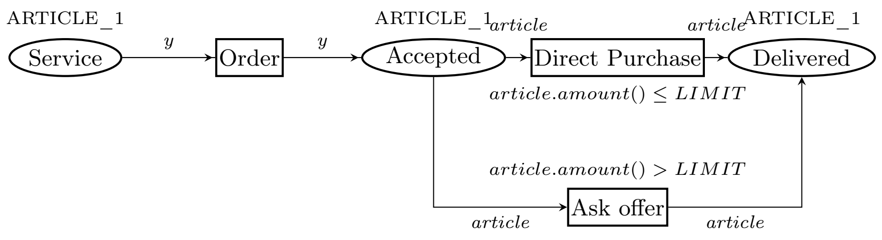
From (8), the abstract marking is obtained by ignoring the subnet marking. Hence, ϕ is linear and total over $\mathbb{M}_r$.
From (6), the firing of an alternative transition corresponds to firing the abstract transition. Hence, ϕ is linear and total over $\mathbb{Y}_r$.
From (6) $\forall Y_r \in \mathbb{Y}_r$, Y_r is complete since only one of the alternative transitions can be fired in a given marking.
From (3), (4) and (7), $\forall Y_r \in \mathbb{Y}_r, \forall M_r \in \mathbb{M}_r$ *si* $M_r \geq E_r^-(Y_r)$ then
$\phi(M_r) \geq \phi(E_r^-)(\phi(Y_r))$ and
$\phi(M_r + E_r^+(Y_r) - E_r^-(Y_r)) = \phi(M_r) + \phi(E_r^+)(\phi(Y_r)) - \phi(E_r^-)(\phi(Y_r))$.
$\phi(M_r + E_r^+(Y_r) - E_r^-(Y_r))$ is the effect after abstraction of the firing of the refined step Y_r from the refined marking M_r and $\phi(M_r) + \phi(E_r^+)(\phi(Y_r)) - \phi(E_r^-)(\phi(Y_r))$ the effect of the firing of the abstract step $\phi(Y_a)$ from the abstract marking $\phi(M_a)$.

Let us consider again the example of Sect. 5.2, with the abstract net in Fig. 6(a). We now want to explicit the accounting rules: if the amount of the purchase is greater than some limit, a call for offers must be issued (see Fig. 6(b)).

Transition *Buy* has been replaced by two alternative transitions: *Direct purchase* et *Ask offer*.



(a) Example abstract net



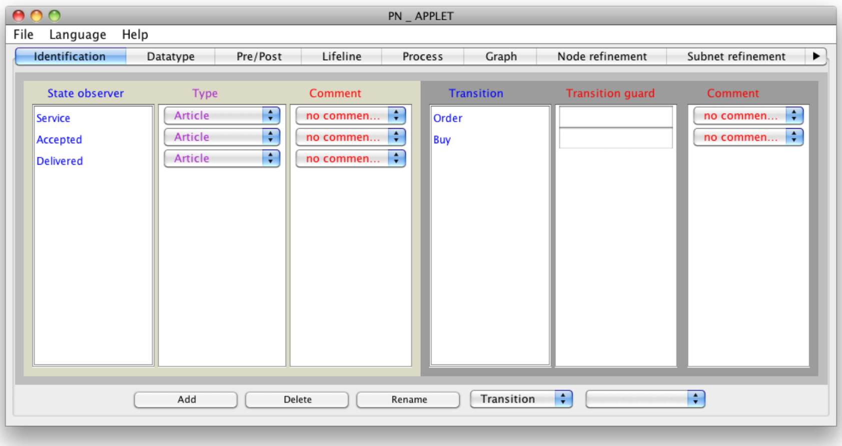
(b) Refined net including the accounting rules

Fig. 6. Refinement of the net model for purchases

7 CPN Refiner: Tool Support for Modelling

We designed CPN Refiner so as to support the development of coloured Petri nets following the approach proposed in [3] and the refinement techniques proposed in this paper.

Let us recall that the Petri net development method presented in [3] proposes the following steps: (i) analyse the text describing the problem and extract the



State observer	Type	Comment
Service	Article	no commen...
Accepted	Article	no commen...
Delivered	Article	no commen...

Transition	Transition guard	Comment
Order		no commen...
Buy		no commen...

Fig. 7. Workshop example: state observers and transitions

events (yielding a state change), state observers, data types, and possibly the modules, (ii) establish the system properties (in particular pre and postconditions of events), (iii) build the coloured Petri net, (iv) check the resulting net properties (some are built-in, but others can be model checked), and update it if necessary.

CPN Refiner supports both the coloured Petri net development given the (typed) state observers and the events (together with their pre and postconditions), and its refinement. CPN Refiner supports node refinement and subnet refinement according to the principles stated in this paper. Type refinement is supported via a mere type modification.

Figure 7 shows the screen where the user entered the state observers and the events for a workshop example, and the generated Petri net is shown in Figure 8. A refined Petri net obtained through alternate transition refinement is shown in Figure 9.

CPN Refiner is build using Java Swing (Eclipse), API JDOM to handle the XML resulting file, library JGraph for the graphical presentation Petri nets.

8 Conclusion and Perspectives

Refinement is used to build a more detailed model (the refined model) from an abstract model. Several refinement notions have been proposed for different aims and various languages, and the work of Lakos deals with refinement for coloured Petri nets. An important point he states is that a model R is a refinement of a model A if for each behaviour of R there is a corresponding behaviour of A. More specifically, he defines three kinds of refinement, type refinement, node

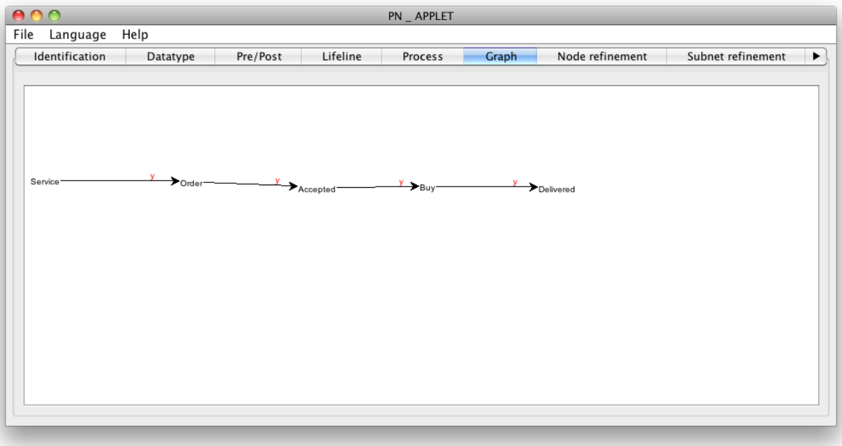


Fig. 8. Workshop example: coloured Petri net

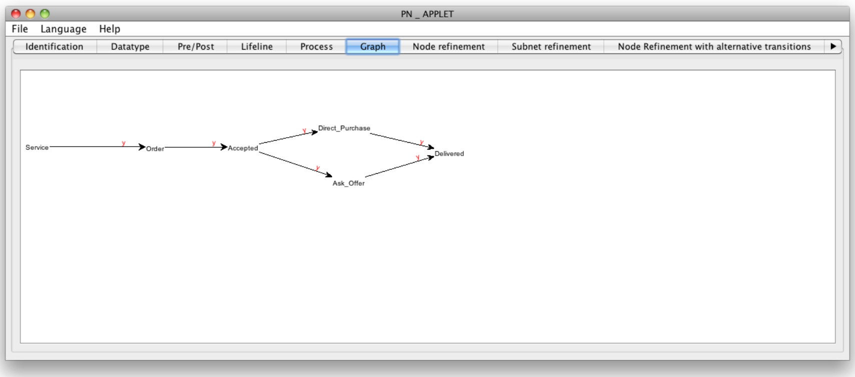


Fig. 9. Workshop example: refined Petri net

refinement (for places or transitions), and subnet refinement, so as to insure this behaviour correspondence.

In this paper, we extend the type refinement and the node refinement proposed by Lakos. For type refinement, we propose two constraints. The subtype relation as defined by Liskov and Wing that should exist between the refined and the abstract type, and Lakos' refinement relation between the refined and the abstract model. We consider four operations on types, that are to add a component, to fix a component value, to add a method, and to modify a method. We studied the conditions that ensure that these operations guarantee both Liskov and Wing subtyping relation and Lakos refinement relation. We also extend the node refinement with a new rule for transitions refined by alternate transitions. This new rule complies with Lakos' refinement principle.

A model development method was proposed for coloured Petri nets in [3]. We developed a tool, CPN-Refiner, for model development of coloured Petri nets following this method, and integrated in this tool the refinement techniques presented here [12]. In the future, we plan to develop large case studies using our method and our tool. We also plan to work on property refinement. Interface with other tools like CPN Tools [1] or CosyVerif [4] is also subject of future work, which should be eased since CPN Refiner uses PNML [2].

References

1. CPN Tools Homepage. `http://cpntools.org/`.
2. PNML reference site. `http://pnml.org/`.
3. C. Choppy, L. Petrucci, and G. Reggio. Designing coloured Petri net models: a method. In *Proc. Workshop on Practical Use of Coloured Petri Nets*, 2007.
4. CosyVerif group, The. CosyVerif home page. `http://cosyverif.org`.
5. Charles Lakos. On the abstraction of coloured Petri nets. In *18th Int. Conference on Application and Theory of Petri Nets ICATPN '97*, volume 1248 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1997.
6. Charles Lakos. Composing abstractions of coloured Petri nets. In Nielsen, M. and Simpson, D., editors, *21st Int. Conf. on Application and Theory of Petri Nets (ICATPN)*, volume 1825 of *Lecture Notes in Computer Science*, pages 323–345. Springer-Verlag, 2000.
7. Charles Lakos and Glenn Lewis. A catalogue of incremental changes for coloured Petri nets. Technical report, Department of Computer Science, University of Adelaide, 1999.
8. Glenn Lewis. *Incremental specification and analysis in the context of coloured Petri nets*. PhD thesis, University of Hobart, Tasmania, 2002.
9. Barbara Liskov and Jeannette M. Wing. Family values: A semantic notion of subtyping. Technical report, Pittsburgh, PA, USA, 1992.
10. Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 118–141, London, UK, 1993. Springer-Verlag.
11. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841, 1994.
12. Alfred Sanogo. *Raffinement des réseaux de Petri colorés*. PhD thesis, Université Paris 13, 2012.