International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:4, 2008

Representation of Coloured Petri Net in Abductive Logic Programming (CPN-LP) and Its Application in Modeling an Intelligent Agent

T. H. Fung

***Abstract*—**Coloured Petri net (**CPN**) has been widely adopted in various areas in Computer Science, including protocol specification, performance evaluation, distributed systems and coordination in multi-agent systems. It provides a graphical representation of a system and has a strong mathematical foundation for proving various properties. This paper proposes a novel representation of a coloured Petri net using an extension of logic programming called abductive logic programming (**ALP**), which is purely based on classical logic. Under such a representation, an implementation of a CPN could be directly obtained, in which every inference step could be treated as a kind of equivalence preserved transformation. We would describe how to implement a CPN under such a representation using common meta-programming techniques in Prolog. We call our framework **CPN-LP** and illustrate its applications in modeling an intelligent agent.

***Keywords*—**Abduction, coloured Petri net, intelligent agent, logic programming.

I. Introduction

IN recent years, agent oriented computing becomes one of the dominant trends of development in Computer Science. Various approaches have been proposed for solving problems in this field. An important area of research concerns communication and coordination amongst different autonomous agents. Coloured Petri net, which was widely used in studying coordination and concurrency in distributed systems [1, 2], has been applied to this research area in multi-agent systems [3, 4]. The advantages of using CPN are that a graphical representation of a system is provided and analytical methods are available for proving various properties, like place-invariant and deadlock. In parallel, another kind of approach based on logic programming (LP) has also been advocated. Logic programming has been extended in various directions. One of the extensions is called abductive logic programming [5]. A number of researches have proposed the use of ALP in multi-agent systems [6, 7]. The advantages of using logic programming are that logical semantics and inference procedure are available for specifying and implementing a multi-agent system.

Manuscript received in July 18, 2006.

T. H. Fung is Manager – Research and Development in Hong Kong Examinations and Assessment Authority (phone: (852) 2239 2679; fax: (852) 2834 5933; e-mail: thfung@hkeaa.edu.hk).

This paper proposes a novel representation of a CPN in ALP such that advantages of both of these approaches could be resulted. Under such a representation, an implementation of the CPN using meta-programming techniques in Prolog could be directly obtained. Each inference step can be regarded as a kind of equivalence preserved transformation. We would discuss the application of our proposed framework (CPN-LP) using a formulation recently proposed by R.A. Kowalski [8] for modeling an intelligent agent: **Thinking = Logic + Control**, where "Control" concerns the manner in using the inference steps. Following this formulation, we argue that CPN could be nicely used for modeling the control component when designing an intelligent agent in a multi-agent system. Using the logical representation we propose, an implementation could be directly obtained.

The structure of the rest of the paper is as follows. First, we would briefly introduce the formal definition of a CPN and ALP in the coming sections. Second, we outline the way to represent a CPN in ALP and illustrate the steps with a simple example. We show how to implement the representation of a CPN. Then we formally define the inference procedure and discuss its properties. Finally, we mention its applications in modeling an intelligent agent and discuss possible future developments.

II. Coloured Petri Net

In Petri net, the possible states of a system are represented by means of ellipses or circles, which are called **places**. The actions or events, which may occur in the system, are represented by means of rectangles, which are called **transitions**. The net also contains a set of directed arrows called **arcs**. Each arc connects a place with a transition or a transition with a place. The current state of a system is represented by using a number of **tokens**. An arbitrary distribution of tokens on the places is called a **marking**. In coloured Petri net, each token contains values of certain type or called **colour**.

A. Simple Example

In Figure 1, which is borrowed from [2], there are 3 places, namely: *B*, *C*, and *S*. *T2* is a transaction. The marking is that two tokens with values equal to *(p, 0)* are in the place *B* and another

three tokens with values equal to *e* are in the place *S*. Thus a place could contain a multi-set of tokens of certain type or colour. Also, each variable in the figure is also declared with a type.

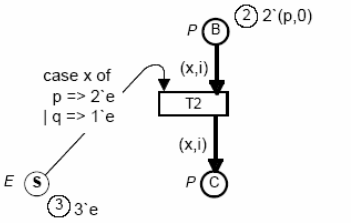

Fig. 1 A simple example of a CPN

Declarations:
type U = with p|q;
type I = int;
*type P = product U*I*
type E = with e;
var x : U;
var i : I;

There are 3 arcs in Figure 1 connecting a place to a transition or a transition to a place. An arc may be inscribed with an arc expression, which results in a multi-set of tokens when evaluating. In the figure, the transition *T2* is enabled and could be fired. When firing the transition, one token of *(p, 0)* and two tokens of *e* would be respectively removed from *B* and *S*, another token of *(p, 0)* would be added to *C*. Sometimes, a transition may be associated with a Boolean expression called a **guard** which has to be evaluated to true when enabling or firing a transition. The formal definitions of these intuitive notions are given in the following.

B. Formal Definitions

Formally, a CPN is defined as a tuple, ***(Σ, P, T, A, N, C, G, E, I)*** where

(i) Σ is called colour sets.
(ii) P is a finite set of places.
(iii) T is a finite set of transitions.
(iv) A is a finite set of arcs.
(v) N is called node function mapping from A into $P \times T \cup T \times P$.
(vi) C is a colour function mapping from P into Σ.
(vii) G is called guard function, mapping a transition in T into a Boolean expression.
(viii) E is called an arc expression function mapping an arc into a multi-set of tokens.
(ix) I is an initialization function mapping a place into a multi-set of tokens.

The following is the formal definitions for a transition being enabled and the resultant marking after firing an enabled transition.

A **binding** of a transition *t*, *(t, b)* is a function *b* defined on variables of *t*, *Var(t)* such that $b(v) \in Type(v)$ for all variables *v* occurring in *t* and $G(t)<b>$ is evaluated to be true (where $G(t)<b>$ stands for the Boolean expression with all the variables in t being replaced with the values as dictated by *b*).

A binding *(t, b)* is **enabled** in a marking *M* if-and-only-if the following property is satisfied.

$\forall p \in P: E(p,t)<b> \leq M(p)$.

Accordingly, the binding, $<T2, \{(x, p), (i, 0)\}>$ is enabled in Figure 1. When a *(t, b)* is **fired** in M_1, the marking is updated to another marking, M_2 as follows.

$\forall p \in P: M_2(p) = (M_1(p) - E(p,t)<b>) + E(t, p)<b>$.

III. Abductive Logic Programming : A Traditional Perspective

Traditionally, abduction means inference to a best explanation and is a pattern of reasoning that occurs in many diverse areas. It proceeds from an outcome to a hypothesis that best explains or accounts for the outcome.

$A \leftarrow B$
A

Conclude *B* as an explanation for the outcome *A*

In general, it can be formulated as follows. Given a set of sentences *T* (a theory or an agent's belief) and a sentence *Q* (an outcome), to a first approximation, the abductive task can be characterized as the problem of finding a set of sentences Δ such that

$T \cup \Delta \models Q$.

Moreover, we need to restrict Δ so that it conveys some reasons why the outcome holds. We want to explain one effect in terms of some causes; instead of other similar effects. Therefore the explanations are often restricted to a special pre-specified and domain-specific class of sentences called **abducibles**. In addition to *T*, **integrity constraints** (also called **propagation rules**), *IC* may be useful to avoid unintended explanations produced. A simple example below is used to illustrate these intuitive notions.

A. Simple Example

T: grass_is_wet ← rain_last_night
grass_is_wet ← sprinker_was_on
no_cloud_last_night
IC: cloudy_last_night ← rain_last_night
false ← cloudy_last_night, no_cloud_last_night
Ab: rain_last_night, sprinker_was_on, cloudy_last_night
Q: grass_is_wet.

Without the use of integrity constraints, *IC*, the possible explanation for the outcome, *Q*; i.e. *grass_is_wet* could be explained by either *rain_last_night* or *sprinker_was_on*. In the

presence of *IC*, the explanation *rain_last_night* would deduce falsity and thus unacceptable. In general, a further criterion is added when constructing an explanation; i.e.

$T \cup \Delta \models IC$.

It should be noted that from *Q* to obtain one of the explanations, backward reasoning is employed. On the other hand, from one of the plausible explanations, *rain_last_night* to obtain the falsity, forward reasoning is employed. Thus in abductive reasoning, **backward reasoning** and **forward reasoning** are integrated together.

B. *Formal Definitions*

Similarly, a ALP is defined as a tuple, (***T, IC, Ab, ThC***)

(i) *T* is called the definitions for defined predicates (in the form of usual Prolog clauses); i.e. those predicates which are not abducible predicates and built-in predicates.

(ii) *IC* is the set of integrity constraints or propagation rules.

(iii) *Ab* is the set of abducible predicates; i.e. predicates without any definitions.

(iv) *ThC* is some theory in form of logical formulae; but in general, would not be stated out explicitly. It is mainly used for justifying those operations when handling built-in predicates. For example, for handling ≠, =, *ThC* includes the **Clark equality theory (CET)** [9] which is implemented in Prolog using standard unification algorithm.

In this paper, *ThC* is also used for logically justifying the steps when modeling the removal of tokens after firing a transition. This would be further explained in a latter section.

IV. Representation of CPN in ALP

Superficially, these two formalisms seem to be totally different from each other. Also, destruction of tokens when firing a transition usually lead one to believe that such an operation could not be appropriately modeled using classical logic.

A. *Simple example*

To clarify our approach, we use the CPN in Figure 1 as an illustration.

In the representation we propose, a marking in a CPN is represented as a set of abducible atoms (in the form of *token(ID, Colour, Place)*) with abducible, *Ab*={*token*}. For the marking of CPN in Figure 1, it is represented as

{ token(id1, col(p,0), b), token(id2, col(p,0), b),
token(id3, col(e), s), token(id4, col(e), s),
token(id5, col(e),s)}.

Here, we adopt the conventions in Prolog. Names starting with a small letter stand for constants, while those starting with a capital letter stand for variables. A (set of) propagation rule(s) (in the form of *trans(ID, ListOfInputs)* ← *List of tokens requested*) would be responsible for firing a transition in a CPN. For *T2* in Figure 1, the following rules are used.

trans(t2, [p,I])← token(Id1, col(p, I), b), token(Id2,col(e), s),
token(Id3,col(e),s), Id2 ≠ Id3.
trans(t2, [q,I])← token(Id1, col(p, I), b), token(Id2,col(e), s).

Thus checking whether a transition is enabled under a marking and firing the transition in a CPN is just a kind of forward reasoning commonly used in many production systems.

For determining the set of output tokens when firing an enabled transition, the definition of an atom (in the form of *trans(ID, ListOfInputs)←List of tokens generated*) for that transition and backward reasoning are employed. Again, for *T2* in Figure 1, the corresponding definition is as follows.

trans(t2, [X, I])← gensym(id, ID), token(ID, col(X,I), c),
place(c, col(X, I).

Note that *gensym* is a built-in predicate available in most Prolog systems for generating a unique serial number, which should be logically treated as a distinct skolem constant. The atom *place(c, col(X, I))* is used as a safeguard to check the appropriateness of the colour of the newly generated token *token(ID, col(X,I), c)*. The corresponding definition is below.

place(c, col(p, I)) ← integer(I).
place(c, col(q, I)) ← integer(I).

Finally, we define a computation state as a tuple consisting of two sets $S = (S_1, S_2)$. At the beginning, S_1 consists of abducible atoms representing the initial marking a CPN and S_2 is empty. For Figure 1, the computation state, *S* is defined as

S=({ token(id1, col(p,0), b), token(id2, col(p,0), b),
token(id3, col(e), s), token(id4, col(e), s),
token(id5, col(e),s)}, {}).

By forward reasoning with a propagation rule, a transition is fired and the tokens occurring in the body of the propagation rule would be removed from S_1 to S_2 and the head of the rule is added to S_1. Then by backward reasoning with the definition of the transition, the corresponding new token(s) would be added to S_1. This process is repeated. For firing the transition *T2* in Figure 1, the changes could be described as follows.

S=({ token(id1, col(p,0), b), token(id2, col(p,0), b),
token(id3, col(e), s), token(id4, col(e), s),
token(id5, col(e),s)}, {})
forward reasoning
S'=({ trans(t2, [p,I]), token(id2, col(p,0), b),
token(id3, col(e), s) }, { token(id1, col(p,0), b),
token(id4, col(e), s), token(id5, col(e),s)})
backward reasoning
S"=({ token(id6, col(p,0), c), token(id2, col(p,0), b),
token(id3, col(e), s) }, { token(id1, col(p,0), b),
token(id4, col(e), s), token(id5, col(e),s)})

S_1 contains the tokens in the current marking of a CPN and S_2 contains the tokens, which occur in a certain previous state.

B. Comparison

A comparison table is below to highlight the difference between a CPN and the corresponding logical representation we propose.

TABLE I
COMPARISON BETWEEN TWO FORMALISMS: CPN VS. ALP

CPN	ALP
Structure: Node function	Implicit in the propagation rules and definitions for transitions
Place: Name Colour set Initial marking	Use the definitions of predicate *place* and functor, *col*: *place(Place, col(Type)).* Initial state: $S=(S_1, S_2)$ where S_1 is a set of atoms with predicate, *token* and S_2 is empty
Transition: Name Guard	Use the definitions of predicate *trans*: *trans(ID, ListOfInputs)←* *List of tokens generated* Note: Guard is represented in the condition of a propagation rule.
Arc expression: It is evaluated to yield a multi-set of tokens with colour	Represented in the body of a propagation rule for arcs from place to transition *trans(ID, ListOfInputs) ←* *List of tokens requested*
Binding:<*T2,{(x,p), (i,0)}*>	Prolog unification process when implementing the propagation rule
Enable a binding	Enable a propagation rule using an appropriate ground instantiation
Fire a transition: A multi-set tokens is removed from each input place and is added to each output place	Forward reasoning using propagation rule and backward reasoning using a definition in order to update a computation state
Declarations: Types, variables Functions/ operations (for defining an arc expression)	Following common Prolog's conventions: No need These could be easily achieved using clauses for defined predicates

C. Full Example: Resource Allocation

Before we mention the implementation, and formalize the inference steps and discuss its properties, we show a full example borrowing from [2] in Figure 2. Note that the initial distribution of tokens (if any) in each relevant place of a CPN is indicated as a multi-set of tokens with an underline. Initially, there are 11 tokens in the CPN.

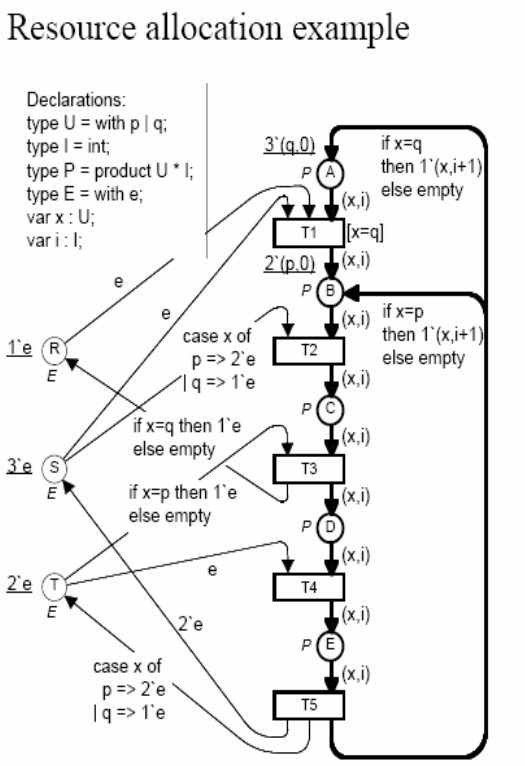

Fig. 2 A full example of CPN

Propagation rules:

trans(t1, [q, I]) ← token(Id1, col(q,I), a), token(Id2,col(e),r),
token(Id3, col(e), s).
trans(t2, [p,I]) ← token(Id1, col(p, I), b),
token(Id2,col(e), s), token(Id3,col(e),s),
Id2 ≠ Id3.
trans(t2, [q,I]) ← token(Id1,col(q, I), b),token(Id2, col(e), s).
trans(t3, [p,I]) ← token(Id1, col(p,I),c), token(Id2, col(e), t).
trans(t3(q,I) ← token(Id2, col(q,I), c).
trans(t4,[P,I]) ← token(Id1, col(P,I), d), token(Id2, col(e), t).
trans(t5,[P,I]) ← token(Id1, col(P,I), e).

Definitions:

place(a, col(X,I)) ← type(X),integer(I).
place(b, col(X,I)) ← type(X),integer(I).
place(c, col(X,I)) ← type(X),integer(I).
place(d, col(X,I)) ← type(X),integer(I).
place(e, col(X,I)) ← type(X),integer(I).
place(r, col(e)). place(s, col(e)).place(t, col(e)).
type(p). type(q).

trans(t1,[P,I])←place(b, col(P,I),gensym(id, ID),
token(ID, col(P,I), b).
trans(t2,[P,I])← place(c, col(P,I)),gensym(id,ID),
token(ID, col(P,I), c).
trans(t3,[p,I])← place(d, col(p,I)),gensym(id,ID),
token(ID, col(p,I), d).

```
trans(t3, [q,I])←place(d, col(q,I)), gensym(id,ID1),
                  token(ID1, col(q,I), d), gensym(id, ID2),
                  token(ID2, col(e), r).
trans(t4,[P, I])←place(e, col(P,I)), gensym(id, ID),
                  token(ID, col(P,I), e).
trans(t5,[p,I])← place(b, col(p,I)),gensym(id, ID1),
   I1 is I + 1, token(ID1, col(p,I1),b), gensym(id, ID2),
   gensym(id,ID3), token(ID2, col(e),s),
    token(ID3, col(e), s),  gensym(id, ID4), gensym(id,ID5),
   token(ID4, col(e),t), token(ID5, col(e), t).
trans(t5, [q,I])←place(a, col(q,I)), gensym(id, ID1),
   I1 is I + 1, token(ID1, col(p,I1),a),
   gensym(id, ID2),gensym(id,ID3),token(ID2, col(e),s),
    token(ID3, col(e), s),  gensym(id, ID4),
    token(ID4, col(e),t).
```

Initial computation state:

```
S=({token(id1,col(q,0),a), token(id2, col(q,0),a),
      token(id3,col(q,0),a),token(id4,col(p,0),b),
      token(id5,col(p,0),b),token(id6, col(e),r),
      token(id7,col(e),s),token(id8,col(e),s), token(id9,col(e),s),
      token(id10, col(e),t),
       token(id11, col(e),t)}, {})
```

V. Implementation

In this section, we briefly sketch how to implement a CPN using the logical representation proposed above. It is largely in line with the format shown in the previous section with some modifications for efficiency of programming.

We adopt SWI-Prolog [10] for the implementation of a CPN using the meta-programming techniques. To ease the programming effort for checking whether a transition is enabled and firing a transition, each transition rule is represented using the following format.

```
progRule(ID, DefinedAtomAdded,
              body(ListOfTokens, Constraints, ListOfVars) )
```

For each propagation rule, we give a unique number, *ID* for identification purpose. The tokens required for firing the rule is represented using the functor *body*, which has three arguments. The first argument is the list of tokens required. The second argument is the constraints on the variables occurring in the tokens required. The third argument is the list of the variables occurring in the tokens. For example, the second propagation rule for the CPN in Figure 2 is represented as follows.

```
progRule(r2, trans( t2,[p,I], LstOfTokensGenerated),
     body([token(Id1, col(p, I), b), token( Id2,col(e), s),
    token(Id3,col(e),s)], [not(Id2 =Id3)],[Id1,I,Id2,Id3]) ).
```

Based on such a format, all the rules and the corresponding instantiations that are enabled in a computation state can be found using the following program clauses and the built-in meta-predicate *setof*.

```
setof( c(RID,LV),
   enableRule(RID, LV, TokensInCurrentState),
   ListRuleVars)
```

where *enableRule* is defined as

```
    enableRule(RID, LV, LstTokens):-
            progRule(RID, _, body(LT, LC, LV) ),
            enable(LT, LC, LV, LstTokens).
    enable(LT, LC, LV, LstTokens) :-
            subList(LT, LstTokens),
            fulfillAll(LC).
```

Simply speaking, a rule is enabled when all the tokens could be found in the current state (*subList(LT, LstTokens)*), and the corresponding constraints on the variables are fulfilled (*fulfillAll(LC)*).

After finding all enabled propagation rules, we randomly choose one to fire; i.e. the head of the propagation rule is added to the computation state. Next, we mention how to handle a defined atom. Each defined atom would have a (set of) clause(s) as its definition. For atoms of predicate *trans*, their definitions are in the following format.

```
trans(ID, ListOfInputVars, ListOfTokensGen) :-
                                    B1,B2,….., Bn.
```

As an illustration, the definition for the transition *T5* would be as follows.

```
trans(t5,[p,I], [token(ID2, col(e),s), token(ID3, col(e), s),
        token(ID1, col(p,I1), b), token(ID4, col(e), t),
        token(ID5, col(e), t)] ):-
   place(b, col(p,I)), gensym(id, ID1), I1 is I + 1,
   gensym(id, ID2), gensym(id,ID3) ,gensym(id, ID4),
   gensym(id,ID5).
trans(t5,[q,I], [token(ID1, col(q,I1),a), token(ID2, col(e),s),
            token(ID3, col(e), s), token(ID4, col(e),t)]):-
   place(a, col(q,I)), gensym(id, ID1), I1 is I + 1,
   gensym(id, ID2),gensym(id,ID3), gensym(id, ID4).
```

The third argument, *ListOfTokensGen* is the list of tokens that would be generated. The body of the clause, *B1*,*B2*,…..,*Bn* is used to determine the ground values of the variables occurring in these tokens based on the values of the input variables.

The cycle of the reasoning process is established using the predicate *abdemo* as shown below.

```
abdemo(state(prc([],[HTok|Toks]),Ls) ) :-
     setof( c(RID,LV),
            enableRule(RID, LV, [HTok|Toks]), ListRuleVars),
     ListRuleVars=[HRule|TRules],!,
     selectOne(ListRuleVars, RuleVars),
     fireRule(RuleVars, [HTok|Toks], Ls, LH1, LTok1, Ls1),
     abdemo( state(prc(LH1, LTok1), Ls1) ).
abdemo(state(prc([trans(ID,ListIn,ListTokens)|Ld],Lt),
        Ls) ) :-
     call(trans(ID, ListIn, ListTokens)),!,
     append(ListTokens, Lt, Lt1),
     abdemo(state(prc(Ld, Lt1), Ls)).
abdemo( state(prc([],Lt), Ls) ):-
     !, write('End of the token game ').
```

Note that a computation state is represented as an atom

```
    state( prc(ListofDefinedAtoms, ListofCurrentTokens),
                                Listof PreviousTokens ).
```

The first argument of the functor, *prc*, *ListofDefinedAtoms* is the list of defined atoms with predicates equal to *trans*. The second argument *ListofCurrentTokens* is the list of current ground tokens which reflect the current system state. Finally,

the second argument, *Listof PreviousTokens* of functor, *state* is the list of previous tokens.

The initial marking would be represented using a number of atoms (in the form of *initial(token(ID,Colour,Place)).* For the CPN in Figure 2, the corresponding clauses are shown below.

```
initial(token(ID,col(q,0),a)):-gensym(id, ID). % ID=id1
initial(token(ID, col(q,0),a)):-gensym(id, ID). % ID=id2
initial(token(ID,col(q,0),a)):-gensym(id, ID). % ID=id3
initial(token(ID,col(p,0),b)):-gensym(id, ID). % ID=id4
initial(token(ID,col(p,0),b)):-gensym(id, ID). % ID=id5
initial(token(ID, col(e),r)):-gensym(id, ID). % ID=id6
initial(token(ID,col(e),s)):-gensym(id, ID). % ID=id7
initial(token(ID,col(e),s)):-gensym(id, ID). % ID=id8
initial(token(ID,col(e),s)):-gensym(id, ID). % ID=id9
initial(token(ID, col(e),t)):-gensym(id, ID). % ID=id10
initial(token(ID,col(e),t)):-gensym(id, ID). % ID=id11
```

The whole process is started by using the following clause.

```
start:-
   findall(T, initial(T), ListOfInitialTokens),
   abdemo(state( prc([],ListOfInitialTokens), [])).
```

The output of firing first two rules when running the logical representation of the CPN in Figure 2 is shown below.

Firing the rule, *r1* (i.e. *T1*) using tokens of *id1*, *id6* and *id7*, the list of current tokens is changed to:

```
token(id12, col(q, 0), b)
token(id2, col(q, 0), a)
token(id3, col(q, 0), a)
token(id4, col(p, 0), b)
token(id5, col(p, 0), b)
token(id8, col(e), s)
token(id9, col(e), s)
token(id10, col(e), t)
token(id11, col(e), t)
```

Firing the rule, *r2* (i.e. *T2* with *x=p*) using tokens of *id5*, *id9*, *id8*, the list of current tokens is changed to:

```
token(id13, col(p, 0), c)
token(id12, col(q, 0), b)
token(id2, col(q, 0), a)
token(id3, col(q, 0), a)
token(id4, col(p, 0), b)
token(id10, col(e), t)
token(id11, col(e), t)
```

VI. Proof Procedure And Its Properties

At the introduction, we claim that each inference step can be regarded as an equivalence preserved transformation. Before we prove such a property, we introduce some definitions and formally define the inference steps below. Note that a conjunction of atoms (or negated atoms) and a set of atoms (or negated atoms) are used interchangeably.

Definition 1: Given an abductive logic program, $P=<T, IC, Ab, ThC>$, the semantics of the program, ***Sem(P)*** is defined as

$$Sem(P) \equiv Comp(T) \cup IC \cup ThC$$

where *Comp(T)* is the Clark completion semantics [9] applied to the (user-) defined predicates. □

Simply speaking, under Clark completion semantics, a general logic program is a set of if-and-only-if definitions. For example, we have a number of clauses for a predicate, say *p*.

$$p(\boldsymbol{t}) \leftarrow \boldsymbol{B_1}$$
$$p(\boldsymbol{t}) \leftarrow \boldsymbol{B_2}$$
$$\ldots\ldots\ldots\ldots$$
$$p(\boldsymbol{t}) \leftarrow \boldsymbol{B_n}$$

Under the Clark completion semantics, these clauses mean the following if-and-only-if logical statement.

$$p(\boldsymbol{t}) \leftrightarrow \boldsymbol{B_1} \vee \boldsymbol{B_2} \vee \ldots\ldots \vee \boldsymbol{B_n}$$

Also, in this paper, *Ab* would be simply equal to {*token*}.

Definition 2: A **computation state, *S*** is a tuple of 2 sets of ground atoms whose predicate could be either of *trans* or *token*; i.e. $\boldsymbol{S=(S_1, S_2)}$. An **initial computation state** $\boldsymbol{S_0}$ is equal to $\boldsymbol{(S_1, \{\,\})}$. □

Definition 3 (Propagation rule):
A propagation rule in *IC* is in the form of

$$trans(\boldsymbol{t}) \leftarrow \boldsymbol{Ts}, \boldsymbol{C}.$$

All the variables in ***t*** and ***C*** must occur in ***Ts***, which stands for a conjunction of atoms with predicates equal to *token*. The atoms or negated atoms in ***C*** would be of some built-in predicates.

□

Definition 4: A **propagation step**: Given an abductive logic program, *P* a computation state, $S=(S_1, S_2)$, a propagation rule, $trans(\boldsymbol{t}) \leftarrow \boldsymbol{Ts}, \boldsymbol{C}$ and a ground instantiation σ for the variables in ***Ts***, a propagation step is defined as follows: If $\boldsymbol{Ts}.\sigma \subseteq S_1$,

$$Sem(P) \models \boldsymbol{C}.\sigma$$

Then the next computation state is $S'=(S'_1, S'_2)$ with

$$S'_1 = (S_1 - \boldsymbol{Ts}.\sigma) \cup trans(\boldsymbol{t}).\sigma,$$
$$S'_2 = S_2 \cup \boldsymbol{Ts}.\sigma.$$ □

Note that since the S_1 contains only ground atoms and all variables in ***t*** must occur in ***Ts***, $trans(\boldsymbol{t}).\sigma$ must be ground.

Definition 5 (Definition clause):
(i) A definition clause in *T* whose head is of predicate *trans* is in the form of

$$trans(\boldsymbol{t}) \leftarrow \boldsymbol{B}, \boldsymbol{Ts}$$

where ***Ts*** is a conjunction of atoms of predicate *token* whose variables must occur either in ***t*** or in ***B***. And ***B*** is a conjunction of atoms or negated atoms whose predicates are either built-in or (user-) defined but not *trans*. The variables in negated atoms must occur in ***t*** or atoms in ***B***.

(ii) A definition clause in *T* whose head is of other predicates; instead of *trans* is in the form of

$$p(\boldsymbol{t}) \leftarrow \boldsymbol{B}$$

where ***B*** could be empty or is a conjunction of atoms or negated atoms whose predicates would **not** be of *token* or *trans* and the variables in ***t*** must occur in some atoms in ***B***. Also, the variables in negated atoms must occur in ***t*** or atoms in ***B***. □

Definition 6: An **unfolding step**: given a definition clause $trans(\boldsymbol{t}) \leftarrow \boldsymbol{B}, \boldsymbol{Ts}$ in *T* of an abductive logic program, *P* and a computation state $S=(S_1, S_2)$, an unfolding step is defined as

follows:

If $trans(\mathbf{t}).\sigma \in S_1$,
$Sem(P) \models trans(\mathbf{t}).\sigma \leftarrow (\mathbf{B}, \mathbf{Ts}).\sigma.\rho$
where σ is any ground instantiation for variables in $\mathbf{t}$,
ρ is any ground instantiation for variables in $\mathbf{B}$,
then the next computation state is $S'=(S'_1, S'_2)$ with
$S'_1 = (S_1 \cup \mathbf{Ts}.\sigma.\rho) - trans(\mathbf{t}).\sigma$,
$S'_2 = S_2$. □

Note that since the variables in $\mathbf{Ts}$ must occur either in $\mathbf{t}$ or in $\mathbf{B}$, $\mathbf{Ts}.\sigma.\rho$ must be ground.

To ensure that any SLD-NF derivation for establishing the validity of $\mathbf{B}$ in a definition clause, $trans(\mathbf{t}) \leftarrow \mathbf{B}, \mathbf{Ts}$ would be terminated within a finite amount of time, we impose the restriction that T has to be **acyclic** (please refer to [11] for details concerning the termination of an acyclic logic program). Since we request that any definition clause (whose head is of other predicates; instead of *trans*), $p(\mathbf{t}) \leftarrow \mathbf{B}$, the variables in $\mathbf{t}$ must occur in some atoms in $\mathbf{B}$, a ground instantiation of $\mathbf{t}$ would be obtained in any finite successful SLD-NF derivation for $p(\mathbf{t})$.

To facilitate the presentation, we introduce the following notation as a short hand.

Definition 7(An answer): Given an abductive logic program, P and an atom, $trans(\mathbf{t})$, $answer_p(trans(\mathbf{t}).\sigma)$ (where σ is any ground instantiation for variables in $\mathbf{t}$) denotes the set of ground abducible atoms $\mathbf{Ts}.\sigma.\rho$ such that the following holds.

$Sem(P) \models trans(\mathbf{t}).\sigma \leftarrow (\mathbf{B}, \mathbf{Ts}).\sigma.\rho$

where ρ is a ground instantiation for variables in $\mathbf{B}$. □

If P is obvious in the context, the subscript would be omitted.

Definition 8 (A derivation): A derivation starting from an initial computation state, S_0 is defined as a chain of computation states.

$S_0 \rightarrow S' \rightarrow S'' \rightarrow \ldots$

where the next state is derived from the current state by applying either a propagation step or an unfolding step. □

We have formally defined the form of T, IC, Ab and inference steps. Now we have to define ThC. As mentioned above, ThC includes CET, which is stated below in the form of propagation rules.

(1) $f(y_1,\ldots,y_n)=f(z_1,\ldots,z_n) \rightarrow y_1=z_1,\ldots,y_n=z_n$
(2) $f(y_1,\ldots,y_n)=g(z_1,\ldots,z_m), f \neq g \rightarrow false$
(3) y *occurs in* $t, y=t \rightarrow false$

This set of rules justifies the use of unification algorithm for catering the equality. Similarly, ThC also includes the pre-conditions and post-conditions of built-in predicates in the form of propagation rule (**b**uilt-**i**n predicates **t**heory, **BIT**) to justify use of these predicates. For example, for the built-in predicate *integer(I)*, the corresponding rules could be:

$ground(I), I \in INTEGER \rightarrow integer(I) \equiv true$
$ground(I), I \notin INTEGER \rightarrow integer(I) \equiv false$

where *INTEGER* stands for the integer type;
ground(I) is evaluated to true when I is without any variables; otherwise false.

Besides, to capture the characteristics of the execution of a CPN, we add the following two theories to ThC.

Definition 9 (Theory of Mutually Exclusive): Theory of mutually exclusive for IC of an ALP is as follows.

For any two rules:
$trans(\mathbf{t1}) \leftarrow \mathbf{T1s}, \mathbf{C1}$,
$trans(\mathbf{t2}) \leftarrow \mathbf{T2s}, \mathbf{C2}$
in IC, we have
$(\mathbf{T1s}.\sigma 1 \cap \mathbf{T2s}.\sigma 2 \neq \varnothing) \rightarrow$
$((\mathbf{T1s} \wedge \mathbf{C1}).\sigma 1 \wedge (\mathbf{T2s} \wedge \mathbf{C2}).\sigma 2 \equiv false)$.

where $\sigma 1$ is any ground instantiation for variables in $\mathbf{T1s}$ and $\sigma 2$ is any ground instantiation for variables in $\mathbf{T2s}$. □

This theory is to justify the removal of tokens from S_1 to S_2 in a propagation step. It is because according to this theory, after a ground token has been used to fire a selected propagation rule, the body of any ground instances of other propagation rules sharing with the same ground token would be automatically false. Therefore there is no need to consider that ground token again in subsequent steps. Thus the removal of that ground token from S_1 to S_2 is just a kind of housekeeping.

Definition 10 (Theory of Unique Output): Theory of unique output for T of an ALP is as follows.

For any two definition clauses:
$trans(\mathbf{t1}) \leftarrow \mathbf{B1}, \mathbf{T1s}$
$trans(\mathbf{t2}) \leftarrow \mathbf{B2}, \mathbf{T2s}$
in T, we have:
$(trans(\mathbf{t1}).\sigma 1 = trans(\mathbf{t2}).\sigma 2) \rightarrow$
$answer(trans(\mathbf{t1}).\sigma 1) = answer(trans(\mathbf{t1}).\sigma 1)$

where $\sigma 1$ is any ground instantiation for variables in $\mathbf{t1}$; $\sigma 2$ is any ground instantiation for variables in $\mathbf{t2}$. □

This theory ensures that given a ground instance of $trans(\mathbf{t})$ in a S_1 of a computation state, we obtain a set of ground tokens using an unfolding step. According to the theory, this set of ground tokens would be unique. There are no other alternatives.

Now we are ready to prove the properties of the proof procedure.

Theorem 1 (Equivalence Preserved): Given a derivation,
$S_0 \rightarrow S' \rightarrow S'' \rightarrow \ldots$
$Sem(P) \models S' \equiv S''$ □

Proof:
Case (i) S'' is derived from S' using a propagation step: It is trivially true.
Case (ii) S'' is derived from S' using an unfolding step:
Consider the definition clause(s) for an atom, $trans(\mathbf{t})$ in the form of
$trans(\mathbf{t}) \leftarrow Body_1$
$trans(\mathbf{t}) \leftarrow Body_2$
……
$trans(\mathbf{t}) \leftarrow Body_n$
According to $Comp(T)$, we have
$trans(\mathbf{t}) \leftrightarrow Body_1 \vee Body_2 \vee \ldots \vee Body_n$.

However, due to syntactic restriction we impose to T, any SLD-NF derivation starting from a ground version of $trans(\mathbf{t})$

using one of these clause, $trans(t) \leftarrow Body_i$ would result in a ground instantiation for variables in the atoms of *token* predicate in $Body_i$ or a failure within a finite amount of time. Due to theory of unique output, the resultant token set must be unique. Thus we have the ground version of $trans(t) \equiv$ resultant ground tokens. □

Theorem 2 (Satisfaction of Integrity Constraints): Given a finite chain of a derivation

$$S_0 \rightarrow S' \rightarrow S'' \rightarrow \ldots S^*$$

$$Comp(T) \cup S^* \cup ThC \models IC$$ □

Proof:

Due to the exhaustion of application of propagation step and equivalence between a ground version of $trans(t)$ and the resultant ground token obtained by an unfolding step. The result follows immediately. □

VII. Modeling An Intelligent Agent

In [8], R.A. Kowalski proposed the following formulation:

Thinking = Logic + Control

for modeling an intelligent agent. "Control" refers to the manner in which the inference rules of logic are applied. It includes the use of forward and backward reasoning and also the application of inference rules in sequence or in parallel. Under such a formulation, an ALP is proposed to model the agent's different types of thinking using forward and backward reasoning in an agent's observe-think-decide-act cycle:

To cycle,
Observe the world,
Think,
Decide what actions to perform,
Act,
Cycle again.

Following the same line of argument, we further propose that the component "Control" could be modeled using CPN; i.e. the order of the inference steps could be visually and systematically organized with the use of CPN. We use a simple example from [8], which concerns getting help on the London underground in an emergency. In this simple example, perception from the environment could be "there are flames", "there is smoke", "one person attacks another", "someone becomes seriously ill", and "there is an accident". The only candidate action is "press the alarm signal button". Perceptions and actions are represented as abducible atoms. The corresponding CPN is shown in Figure 3. The declarations of the CPN are below:

type P = with flames|smoke|attacks|someone_ill|accident
type E = with emergency
type A = with press_alarm
var x: P
var e: E
var a: A

Based on the CPN in Figure 3, we can write down the corresponding propagation rule as shown below.

trans(processPercept,[P])←
token(Id,col(P),p), place(p, col(P)).
trans(decideAct, [E])←
token(Id, col(E), q), place(q, col(E)).

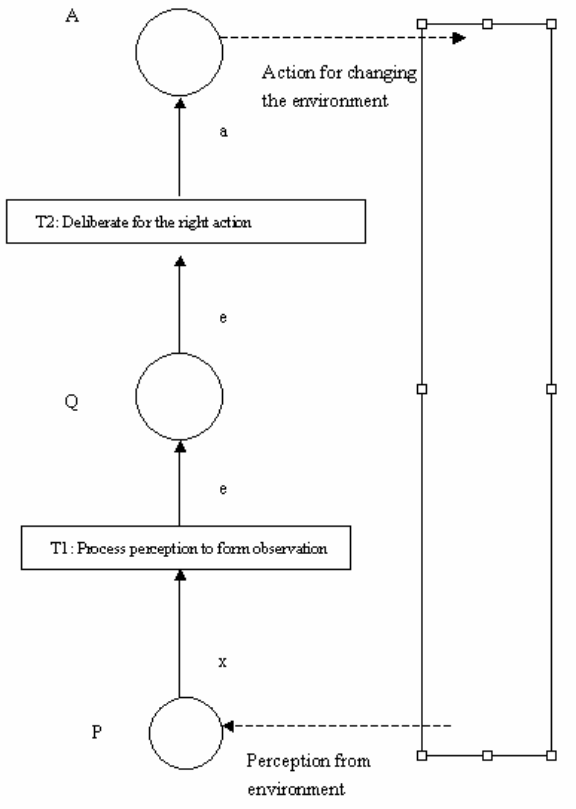

Fig. 3 A CPN for modeling of getting help on the London underground in an emergency

According to the CPN, the definition for the predicate *place* is below.

place(p, col(flames)).
place(p,col(smoke)).
place(p, col(attacks)).
place(p, col(someone_ill)).
place(p, col(accident)).
place(q, col(emergency)).
place(a, col(press_alarm)).

Within the limited scope of perceptions received from the environment, the definition of *processPercept* would be degenerated and simply as follows.

trans(processPercept, [P]) ←
token(ID, col(emergency), q), gensym(passenger, ID).

Thus whenever there is an perception of allowed type from the environment, a token with value equal to *emergency* will be added to the place *Q*. The forward reasoning is classified as a kind of **reactive thinking** of an agent. The newly added token will trigger the second propagation rule. The process of *decideAct* will be relatively more complicated for deciding

right action. The definition would be as follows.

trans(decideAct, [E])←
decideAction(E, Act),
gensym(passenger,ID),
place(a, col(Act)), token(ID, col(Act), a).

where *decideAction* is defined as:

decideAction(emergency, Act) ←
getHelp(Act).
getHelp(Act)←
alert(Person, Act).
alert(driver, press_alarm).

The backward reasoning is classified as a kind of **proactive thinking** of an agent; i.e. try to achieve goals by reducing them to sub-goals. Finally a token with value equal to *press_alarm* would be added to the place *A*, provided that a perception of appropriate type is received from the environment.

To simulate the interaction between an agent and the environment in our implementation, we use the multi-thread utilities provided in SWI-Prolog. Whenever there is a token in the place *A*, the following will be executed.

thread_send_message(envthd, token(ID, col(Act), a))

A message, *token(ID, col(Act), a)* will be sent to the message queue of another thread, called *envthd* which models the environment; while the passenger is simulated using another thread. Within the passenger thread, an infinite loop (which simulates the agent's cycle) could be set up to check any message from the environment using the following clauses.

thread_peek_message(token(ID,Colour,Place)),
thread_get_message(token(ID,Colour,Place)).

For further details of these built-in predicate, please refer to [10].

In this simple example, our implementation of the environment simply echoes the passenger's action. After the environment thread sends a message, say *token(env1, col(attacks),p)* to the passenger thread, the following output would be produced subsequently.

get a message in passenger: token(env1, col(attacks),p)
get a message in environment: Action- col(press_alarm)

In this example, there is only a perception; instead of an effect (*Q*) which is required to be explained in terms of other causes. Thus the aim of the adductive reasoning is to achieve $T \cup \Delta \models IC$ where Δ would be updated due to information from the environment. *IC* becomes the goal we want to achieve or maintain (refer to [8] for details). We argue that one can visualize and systematically organize the order of reactive thinking and proactive thinking with the use of CPN when designing an intelligent agent. A prototype can be developed immediately using the representation of a CPN proposed in this paper. Moreover, when more than one agent involved, their interactions can be easily modeled by passing tokens (abducible atoms) from one to another. Multi-thread utilities provided in SWI-Prolog greatly facilitate the development.

VIII. Conclusion

In this paper, we propose a logical representation for a CPN. As far as we know, this is the first attempt of providing a logical formulation of CPN within the framework of classical logic with well-defined semantics. Moreover, the logical formulation could be executed directly using a meta-interpreter, of which each inference step can be regarded as equivalence preserved transformation. This is the gain on the side of CPN. On the other side, the resultant abductive proof procedure is greatly simplified (as compared with the abductive proof procedure in [5] which has seven inference rules and a very complicated computation state consisting of atoms and rules with existentially or universally quantified variables) using the execution of a CPN as a guideline. There are only two inference steps. The computation state consists of lists of ground atoms with predicate equal to either *token* or *trans*. Such a simplification greatly saves the programming effort and improves implementation efficiency. In view of the wide application of CPN, it is expected that the applicability of the proof procedure would not be greatly scarified. Moreover, CPN provides a visualization of the execution order of forward and backward reasoning. We would like to view that the relation of CPN with ALP is similar to that of DFD (or UML) with conventional (or object-oriented) programming. When designing an intelligent agent, CPN provides a graphical representation, which nicely organizes forward reasoning and backward reasoning in a systematic manner. Based on the graphical representation, a corresponding logic program could be arrived at. One of the difficulties in using logic program as a development or prototyping tool is hard to clarify the execution flow. With the use of CPN, such a difficulty could be largely reduced. We expect that the integration of CPN with logic programming would provide a powerful framework (CPN-LP) for developing multi-agent applications and analyzing their properties.

Future works include the exploration of the framework in various application areas, such as protocol specifications in a multi-agent setting [12,13] and workflow automation, and how the framework could be extended to incorporate the normative positions of an agent, such as obligation, prohibition and permission [14].

Acknowledgment

I would like to express my gratitude to Dr. Choy Sheung On, Steven, who is interested in the proposed framework, and read the draft of this paper and provided valuable comments.

References

[1] K. Jensen, "Coloured Petri Nets: A High-level Language for System Design and Analysis," in *G. Rozenberg(ed.) Advances in Petri Nets 1990, Lecture Notes in Computer Science* Vol. 483, 342-416, Springer-Verlag 1991.

[2] K. Jensen, *Coloured Petri Nets: Basics Concepts, Analysis Methods and Practical Use.* Vol. 1: Basic Concepts, 1992. Vol. 2 : Analysis Methods, 1994. Vol. 3: Practical Use, 1997. Monographs in Theoretical Computer Science, Springer-Verlag.

[3] Daniel Moldt and Frank Wienberg, "Multi-agent Systems based on Coloured Petri Nets," in *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN '97)*, number 1248 in

Lecture Notes in Computer Science, 82-101, Toulouse, France, June 1997.

[4] Jacques Ferber, *Multi-agent Systems An Introduction to Distributed Artificial Intelligence.* English Ed., Pearson Education Ltd., Addison-Wesley, 1999.

[5] T. H. Fung and R. A. Kowalski, "The IFF proof procedure for abductive logic programming," *Journal of Logic Programming*, 33(2): 151-165, November, 1997.

[6] Macro Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni, "Verifiable Agent Interaction in Abductive Logic Programming: The SCIFF proof-procedure", *DEIS Technical Report no. DEIS-LIS-06-001*, Universität delgli Studi di Bologona, March 2006.

[7] Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni, "An Abductive Framework for Information Sharing in Multi-Agent systems," in *Jürgen Dix and João Leite, eds., 4th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-IV),* Fort Lauderdale, FL, January 6-7, 2004. LNAI 3259, 34-52, Springer-Verlag, 2004.

[8] R. A. Kowalski, "The Logical Way to Be Artificially Intelligent". http://www.doc.ic.ac.uk/~rak/ (2002-2006).

[9] K. Doets, *From Logic to Logic Programming*. The M.I.T. Press, Cambridge MA, 1994.

[10] Jan Wielemaker, *SWI-Prolog 5.6 Reference Manual,* updated for version 5.6.10, April 2006. http://www.swi-prolog.org.

[11] K. R. Apt and M. Bezem, "Acyclic Program, " *New Generation Computing*, 29(3): 335-363, 1991.

[12] R. Scott Cost, Yannis Labrou, and Tim Finin, "Coordinating Agents using Agent Communication Languages Conversations", in *Andrea Omicini, Franco Zambonelli, Matthias Klusch, Robert Tolksdorf (eds.) Coordination of Internet Agents Models, Technologies, and Applications*, 183-196, Springer-Verlag, 2001.

[13] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni, "The SOCS Computational Logic Approach to the Specification and Verification of Agent Societies," in *Post-Proceedings of the Global Computing 2004 Workshop (GC 2004)*, Rovereto, Italy, March 9-12, 2004. LNAI 3267, 314-339, Springer-Verlag, 2005.

[14] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, Giovanni Sartor, and Paolo Torroni, "Mapping Deontic Operators to Abductive Expectations," in *Proceedings of 1st International Symposium on Normative Multiagent Systems (NorMAS 2005)*, AISB 2005, Hertfordshire, Hatfield, UK, April 2005.