Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA'06), 2006

Petri Net Plans

Vittorio Amos Ziparo[1] and Luca Iocchi[2]

[1] Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, ziparo@dis.uniroma1.it
[2] Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, iocchi@dis.uniroma1.it

Summary. In this paper we present a novel representation framework based on Petri Nets for describing robot and multi-robot behaviors. The Petri Net Plan (PNP) formalism allows for high level description of complex action interactions that are necessary in programming cognitive robots: non-instantaneous actions, sensing and conditional actions, action failures, concurrent actions, interrupts, action synchronization in a multi-agent context. We show how this framework is capable of describing effective plans for robotic agents which inhabit a dynamic, partially observable and unpredictable environment.

The proposed framework has been implemented and successfully deployed on actual robotic teams in different application scenarios.

1 Introduction

High level programming of mobile robots is very important for developing complex and reliable robotic applications. In this paper we present a framework based on Petri nets that has been used for describing high level robot and multi-robot behaviors. The proposed framework has been used for describing effective plans for robotic agents which inhabit dynamic, partially observable and unpredictable environments, and experimented in different application scenarios, including robotic soccer and rescue competitions.

Our cognitive robots are based on a *heterogeneous hybrid* architecture. These kind of architectures are capable of integrating reactiveness and proactiveness. In particular, they are structured in two layers: a deliberative and an operational one. The former maintains a high level representation of the environment which is used to choose actions; the latter maintains a low level representation which is used to evaluate conditions and to execute basic behaviors (which we call actions). *Hybrid architectures* can further be classified based on how the knowledge is represented. A hybrid architecture may be *homogeneous* if the knowledge is represented in the same way both at the deliberative level and the operational one, *heterogeneous* otherwise. Our approach follows the *heterogeneous* one where the deliberative layer is obtained by specifying high level plans (in fact, the Petri Net Plans that we are describing in this paper), while the operative level maintains numeric information about the state of the robots, integrating different techniques (such as probabilistic localization, dynamic control, etc.).

The objective of this paper is to describe a novel representation framework for high level robot and multi-robot behaviors, its implementation on actual mobile robots, and our experience using such a framework. The proposed framework, called *Petri Net Plans* (PNP), is based on Petri nets [Mur89], a graphical modeling language for dynamic systems.

Our modeling language is one of the many extensions to transition graphs existing in literature. As a difference with such other approaches, e.g. XABSL [LBBJ04], we clearly distinguish action specification and implementation, obtaining a framework which permits easier debugging: first, the semantic is well defined and easily verifiable by automated verification programs; second, we have a high granularity of actions which are grouped by functional properties and physical resources used. Moreover, we provide a rich set of operators for handling complex behaviors.

There exist other languages capable of handling synchronization constraints (e.g., [SA98, PDPW, Fir89]) or knowledge acquisition (e.g., [GL86, Kon97]), but not many which can handle both. One such language is ConGolog [DLL00] which extends Golog for handling concurrent execution but fails in modeling reactive behaviors. For this reason, Golog was further extended introducing interrupts. The resulting language is called RGolog [Rei01]. Our formalism is very rich and includes all of the above mentioned features. It differs from these languages mainly in the way in which the knowledge of the agent is used to represent the properties in the environment and in the higher efficiency of plan execution, due to the absence of computational expensive reasoning procedures during this process. More detailed analysis and comparison with these languages are given in Section 7.

The proposed framework has been implemented and used to control robotic systems in three domains: (i) the RoboCup 4Legged soccer competitions [IN04], (ii) the RoboCup Rescue competitions, and (iii) a multi robot foraging testbed for task assignment experiments based on a token passing approach [FINZ06].

The remainder of the paper is structured as follows: we first define the syntax for our language using Petri nets in terms of operators (i.e., actions) and possible interactions among them. Two types of models for non-instantaneous actions are given:

1. ordinary non-instantaneous actions, which allow complex constructs for action synchronization and failure recovery.
2. sensing non-instantaneous actions, which allow for dynamically sensing properties at execution time and thus for knowledge acquisition.

We then provide a set of operators for handling concurrency, conditionals and iterations. In order to give a clear operational semantics to our modeling language we provide an execution algorithm. After defining what is a correct execution for a plan, we proof that, if a correct execution is possible, then the algorithm will achieve it. The extension of the framework to deal with Multi-Agent planning in provided in Section 5.

Implementation issues are provided in Section 6, while Section 7 contains a discussion about advantages and difficulties in using such a method on our mobile robotic teams, as well as comparison with other approaches. Finally, we conclude the paper by illustrating possible future work.

2 Petri Nets

Petri nets are a graphical and mathematical modeling tool [...] for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic.[Mur89]

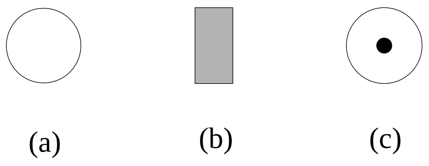

Fig. 1. (a) A place. (b) A Transition. (c) A Place with one token.

Petri nets, as a modeling language, graphically depict the structure of a distributed system as a directed, weighted and bipartite graph. As such, a Petri net has two types of nodes connected by directed weighted arcs (if not labeled we assume a weight of one). The first type is called *place* (Fig. 1a) and may contain zero or more *tokens* (Fig. 1c). The number of tokens in each place (i.e. *marking*) denotes the state of the system.

The other type of nodes, called *transitions* (Fig. 1b), represent the events modeled by the system. Transitions can consume or produce tokens from places according to the rules defining the dynamic behavior of the Petri net (i.e. the firing rule).

More formally, a Petri net can be defined as a tuple

$$PN = < P, T, F, W, M_0 >$$

where:

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *places*.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of *transitions*.
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of edges.
- $W : F \to \{1, 2, 3, \ldots\}$ is a weight function and $w(n_s, n_d)$ denotes the weight of the edge from n_s to n_d.
- $M_0 : P \to \{0, 1, 2, 3, \ldots\}$ is the initial marking.
- $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$

Petri nets are used to model complex systems that can be described in terms of states and their changes. We can define the state changing behavior (i.e. the marking evolution) in a Petri net by the following *firing rule*:

1. A transition t is *enabled*, if each input place p_i (i.e. $(p_i, t) \in F$) is marked with at least $w(p_i, t)$ tokens.
2. An enabled transition may or may not fire, depending on whether related event occurs or not.
3. If an enabled transition t fires, $w(p_i, t)$ tokens are removed for each input place p_i and $w(t, p_o)$ are added to each output place p_o such that $(t, p_o) \in F$.

There exists another type of arc called *inhibitor arc*. This arc is represented as a dashed segment with a small circle (Fig. 7). This connects a place to a transition and enables it when there are no tokens in the place. Obviously no tokens are moved when the transition fires.

Petri Nets with inhibitor arcs are called *Extended Petri Nets*. The use of this connector enables the net to test for the zero and gives to these nets the same modelling power as *Turing machines* [Pet81].

3 Plan Representation

Programming high level behaviors for a mobile robot executing complex tasks in dynamic, partially observable ad unpredictable environments requires a powerful description language.

The reference scenario in this paper is the cognitive control of a four-legged robot (AIBO) involved in robotic soccer. Such complex scenario requires to deal with non-instantaneous actions, sensing and conditional actions, action failures. Moreover, since the AIBO robot can independently move its legs and its head execution of concurrent actions is also needed.

In this section we formally introduce a modeling language for describing robotic behaviors based on Petri nets. The proposed language allows for specifying plans, called *Petri Net Plans (PNP)*, describing complex behaviors of a mobile robot. These plans are defined by combining different kinds of actions (ordinary actions and sensing actions) using control structures, such as if-then-else, while, concurrent execution and interrupts.

A *Petri Net Plan* is a Petri net $< P, T, F, W, M_0 >$ with the following characteristics.

1. Places p_i represent the execution phases of actions; each action α is described by a place corresponding to its initiation (we call it *initial* place of α), one corresponding to its execution (we call it *execution* place of α), and one corresponding to its termination (we call it *termination* place of α);
2. Transitions t_i represent events and are grouped in different categories: action starting transitions, action terminating transitions, action interrupts and control transitions (i.e. transitions that are part of an operator). Transitions may be labeled with conditions that control their firing.
3. $w(f_i, f_j) = 1$, for each $(f_i, f_j) \in F$.
4. M_0 is the initial marking representing a description of the initial state of the robot.

In the following we will focus on the structure of a PNP (i.e. considering only the terms $< P, T, F >$).

A Petri Net Plan is formally defined by a set of elementary structures (i.e. *no-action*, *ordinary action*, *sensing action*) and constructs for combining PNP (i.e. sequences, loops, concurrent execution, interrupts).

Elementary structures.

Elementary PNPs are defined as follows:

1. **no-action** is a PNP defined by a single place and no transitions, i.e. $< \{p_0\}, \emptyset, \emptyset >$ (see Fig.1a), where p_0 is both an initial and a terminating place.

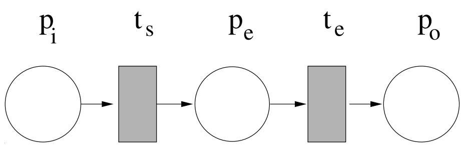

Fig. 2. An ordinary non-instantaneous action.

2. **ordinary-action** is a PNP defined by 3 places and 2 transitions (see Fig. 2):

$$< \{p_i, p_o, p_e\}, \{t_s, t_e\}, \{(p_i, t_s),$$
$$(t_s, p_e), (p_e, t_e), (t_e, p_o)\}\} >$$

   where:
   - p_i is the initial place.
   - p_o is the terminating place.
   - p_e is the execution place.
   - t_s the transition starting the action.
   - t_e the transition terminating the action.

   In order to model those actions which may be considered instantaneous, we introduce the instantaneous variant of the above PNP: $< \{p_i, p_o\}, \{t_a\}, \{(p_i, t_a), (t_a, p_o)\}\} >$ where t_a is the transition representing the event of executing an instantaneous action.
3. **sensing-action** is a PNP defined by places and transitions as described in Fig. 3:

$$< \{p_i, p_e, p_{o_t}, p_{o_f}\}, \{t_s, t_{e_t}, t_{e_f}\}, \{(p_i, t_s),$$
$$(t_s, p_e), (p_e, t_{e_t}), (p_e, t_{e_f}), (t_{e_t}, p_{o_t}), (t_{e_f}, p_{o_f})\} >$$

   where transitions and places are the same of the previous example except for:
   - t_{e_t} and t_{e_f} are, respectively, the transitions ending the action when the sensed property is true and when it is false.
   - p_{o_f} and p_{o_f} are, respectively, the places terminating the the action when the sensed property is true and when it is false.

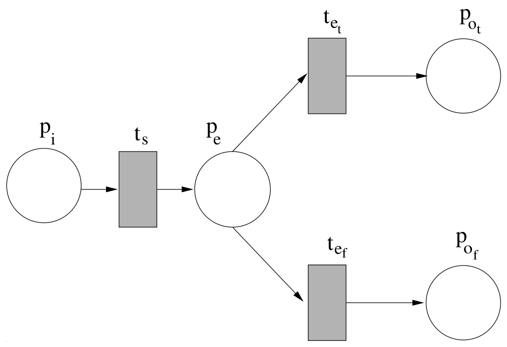

Fig. 3. An non-instantaneous sensing action.

As for the ordinary-action, we define the instantaneous variant of the sensing-action as: $< \{p_i, p_{o_t}, p_{o_f}\}, \{t_{e_t}, t_{e_f}\}, \{(p_i, t_{e_t}), (p_i, t_{e_f}), (t_{e_t}, p_{o_t}), (t_{e_f}, p_{o_f})\} >$.

Operators.

Elementary PNPs can be combined by using the operators sequence, conditional, loops, concurrent execution and interrupts.

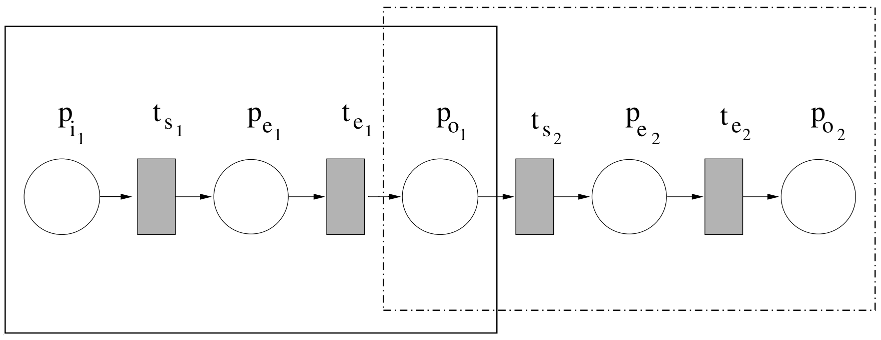

Fig. 4. Sequence of two PNPs.

The *sequence* of two PNPs is defined as follows: given two PNPs $\Gamma_1 =< P_1, T_1, F_1 >$, $\Gamma_2 =< P_2, T_2, F_2 >$ and two places $p_{o_1} \in P_1$ and $p_{i_2} \in P_2$, such that p_{o_1} is a terminating state for an action α_1 in Γ_1 and p_{i_2} is an initial state for an action α_2 in Γ_2, a new PNP $\Gamma =< P, T, F >$ is obtained by joining the places p_{o_1} and p_{i_2} as follows: (i) $P'_2 = P_2 \backslash \{p_{i_2}\}$, is the set of places excluding p_{i_2}, (ii) $\tau(p_{i_2}) = \{t_i | (p_{i_2}, t_i) \in F_2\}$ is the set of transitions following the place p_{i_2}, (iii) $F'_2 = F_2 \backslash \{(p_{i_2}, t') | t' \in \tau(p_{i_2})\}$ is the set of edges of Γ_2 excluding the ones coming from p_{i_2}, (iv) $F'_1 = F_1 \cup \{(p_{o_1}, t') | t' \in \tau(p_{i_2})\}$ is the set of edges of Γ_1 augmented by those obtained connecting the place p_{o_1} to the successors of p_{i_2}, (v) $P = P_1 \cup P'_2$, $T = T_1 \cup T_2$, $F = F_1 \cup F'_2$, is the union of the sets after the above modifications.

The above formulation actually allows for merging two PNPs choosing a terminating place for an action, an initial place for another action and join the two nets making such places to be the same. A graphical representation of this operator is given in Figure 4.

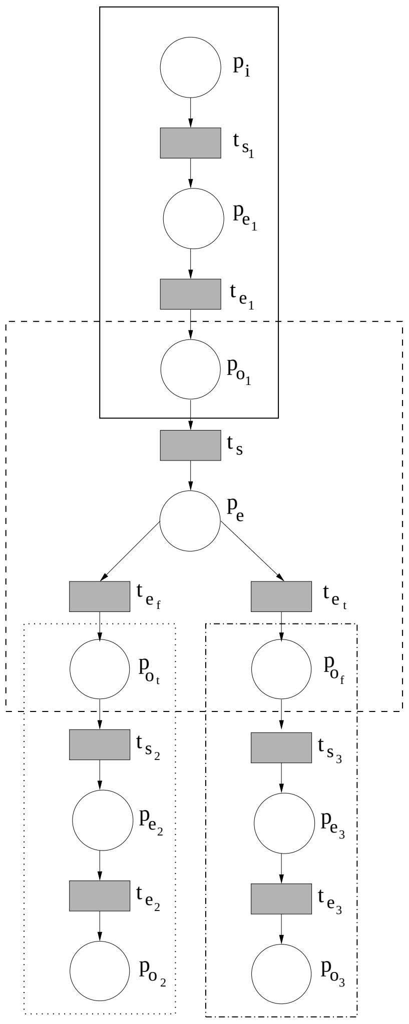

Fig. 5. Conditional structure.

Conditional structures are implemented though sensing actions: given a sensing action α, three PNPs $\mathit{\Gamma}_1$, $\mathit{\Gamma}_2$, $\mathit{\Gamma}_3$, and three places: p_{o_1} a terminating place in $\mathit{\Gamma}_1$, and

p_{i_2}, p_{i_3} initial places in $\mathit{\Gamma}_2$, $\mathit{\Gamma}_3$, a new PNP $\mathit{\Gamma}$ is obtained by joining the initial place of the sensing action α with p_{o_1} and the two terminating places for α with p_{i_2} and p_{i_3}. The joining operation is similar to the one described for the sequence operator and, for maintaining an easy notation, we present it here only in graphical form in Figure 5.

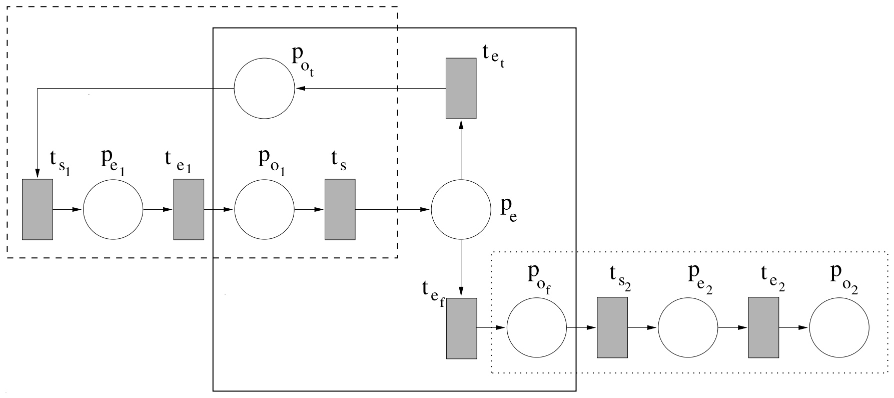

Fig. 6. An *indefinite iteration* which executes the PNP $\mathit{\Gamma}_1$ while the sensed property is true.

Loop structures are also implemented through sensing actions: given a sensing action α, two PNPs $\mathit{\Gamma}_1$, $\mathit{\Gamma}_2$, and three places: p_{o_1} a terminating place in $\mathit{\Gamma}_1$, p_{i_1} an initial place in $\mathit{\Gamma}_1$, p_{i_2} an initial places in $\mathit{\Gamma}_2$, a new PNP is obtained by joining the initial place of the action α with p_{t_1} and the two terminating places for α with p_{i_1} and p_{i_2}. The graphical representation of this operator is given in Figure 6.

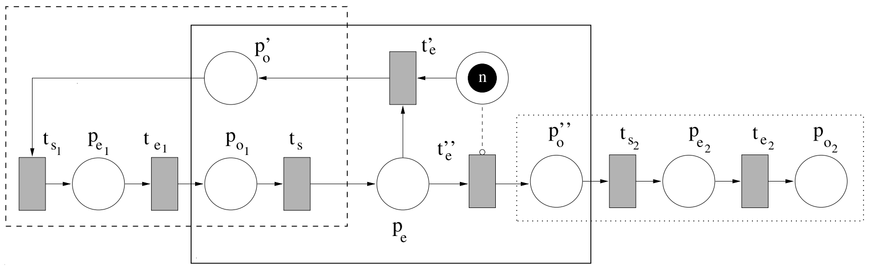

Fig. 7. A *definite iteration* which executes the PNP $\mathit{\Gamma}_1$ $n+1$ times.

Adding to this structure a control place marked with n tokens (Fig. 7), we obtain a definite iteration operator. In this way we can execute $n+1$ times a given net.

Concurrent execution is defined by adding new transitions and edges: given three PNPs $\mathit{\Gamma}_1$, $\mathit{\Gamma}_2$, $\mathit{\Gamma}_3$, a terminating place p_{o_1} in $\mathit{\Gamma}_1$, and two initial places p_{i_2}, p_{i_3}, re-

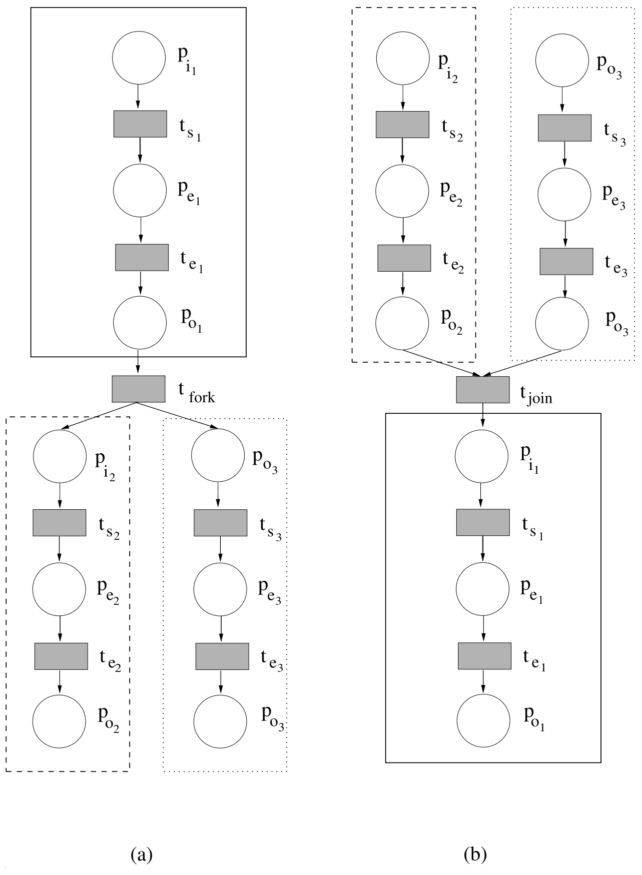

Fig. 8. (a) The fork structure. (b) The join structure.

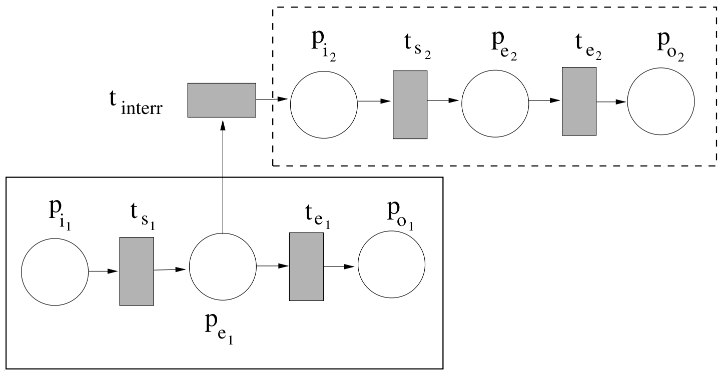

Fig. 9. Interrupt structure where possibly Γ_1 is interrupted and then Γ_2 executed.

spectively in Γ_2, Γ_3, a new PNP is obtained by adding one transitions t_{fork} and three edges (p_{o_1}, t_{fork}), (t_{fork}, p_{i_2}), (t_{fork}, p_{i_3}) to the union of the sets specifying Γ_1, Γ_2, Γ_3. The graphical representation of this operator is given in Figure 8(a).

In a similar way we can define an operator to join concurrent execution: given three PNPs Γ_1, Γ_2, Γ_3, an initial place p_{i_1} in Γ_1, and two terminating places p_{o_2}, p_{o_3}, respectively in Γ_2, Γ_3, a new PNP is obtained by adding one transitions t_{join} and three edges (p_{o_2}, t_{join}), (p_{o_3}, t_{join}), (t_{join}, p_{i_1}) to the union of the sets specifying Γ_1, Γ_2, Γ_3. The graphical representation of this operator is given in Figure 8(b).

Interrupt constructs are defined by adding a new transition and edges to the execution place of an action: given two PNPs Γ_1, Γ_2, an execution place p_{e_1} in Γ_1, an initial place p_{i_2} in Γ_2, a new PNP is obtained by adding a new transition t_{interr} and new edges (p_{e_1}, t_{interr}), (t_{interr}, p_{i_2}) to the union of the sets specifying Γ_1, Γ_2. The graphical representation of this operator is given in Figure 9.

Labeling transitions.

In order to specify external events occurring during task execution, we define a labeling mechanism for transitions in the net. In particular, all transitions may be labeled with conditions which must be verified in order to be fired when enabled. A condition ϕ on the transition t is denoted with $t.\phi$. If no condition is specified for a transition, we will assume that it is the condition $True$. Sometimes it is useful to set such condition to $False$ in the ending transitions to model non-terminating actions. These are usually supporting actions (see for example, the action textttrackBall in the following example) that are executed concurrently with a main action that actually determines plan transitions.

3.1 Example: A simple Robocup 4Legged Striker

We will show a simple plan for a Robocup 4Legged Striker. The following example consists of a model for a robot which must seek for the ball and eventually reach it. We have the following primitive behaviors:

1. `approachBall` which is a behavior for approaching the ball controlling the leg actuators. This action is modeled as a non-instantaneous action.
2. `trackBall` which is a behavior for tracking the movement of the ball with the camera positioned on the robot's head. This action is modeled as a non-instantaneous action.
3. `seekBall` which is a behavior for seeking the ball modeled as a non-instantaneous action.

In Figure 10 we show a plan for this task. The robot seeks for the ball which we assume is not seen. When it finds it, the current state will move to the one where the ball is seen. In this case, the robot will concurrently move the legs to approach the ball and track its position with the camera positioned on the head. When the robot is sufficiently near to the ball, the actions `approachBall` and `trackBall` will terminate their execution at the same time thus reaching the goal state.

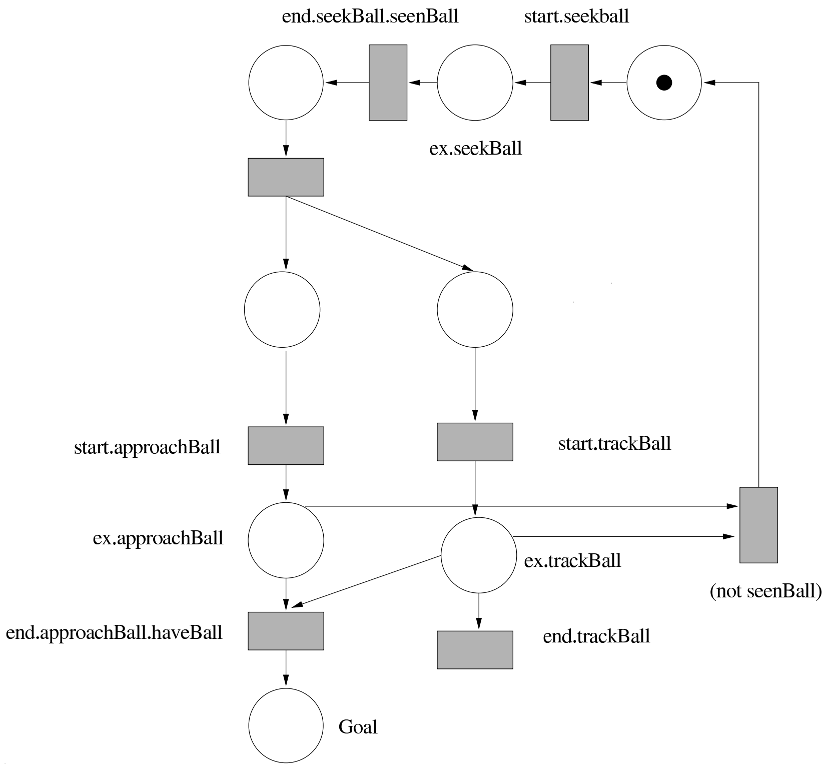

Fig. 10. A simple attacker from the Robocup Soccer domain.

Moreover, if while approaching the ball the robot looses visual contact with the ball, an interrupt will trigger the system to abort the current actions and move to the state where the ball is not seen. This loop will continue until the robot reaches the ball.

4 Plan Semantics

In this section we provide an operational semantics for the execution of PNPs and we present an algorithm that correctly executes a PNP, in the sense that it correctly performs transitions reaching a final state according with the occurrence of external events.

The state of an agent during the execution of a PNP is given by its marking. Transitions between the agent states are thus modelled by transitions in the PNP, i.e. by evolution of its markings. We thus give the definitions for executable transitions of a PNP, that allows for defining the notion of execution of a PNP and of correct execution of a PNP.

During the execution of a plan, and thus during the transitions we are defining, we assume that the robot is provided with a set of functions that are able to evaluate its internal state. These functions are used to evaluate the conditions labelling the transitions of the PNP and thus determine when and how it is possible to perform such transitions.

Definition 1. Possible Transitions in a PNP. *Given two markings M_i, M_{i+1}, a transition from M_i to M_{i+1} is possible iff $\exists t \in T$, such that (i) $\forall p' \in P$, s.t. $(p', t) \in F$, then $M_i(p') > 0$; (ii) $M_{i+1}(p') = M_i(p') - 1$ for each $p' \in P$, s.t. $(p', t) \in F$; (iii) $M_{i+1}(p'') = 1$ for each $p'' \in P$, s.t. $(t, p'') \in F$.*

A possible transition from M_i to M_{i+1} is denoted by $M_i \rightarrow M_{i+1}$.

Definition 2. Executable transition in a PNP. *Given two markings M_i, M_{i+1} and a situation at time τ, a transition from M_i to M_{i+1} in the situation τ is executable iff $\exists t \in T$, such that a transition from M_i to M_{i+1} is possible and the event condition ϕ labelling the transition t (denoted with $t.\phi$) is verified in situation τ.*

An executable transition from M_i to M_{i+1} in situation τ is denoted by $M_i \Rightarrow_\tau M_{i+1}$.

In order to specify the desired states for the system, we introduce the set $GoalMarkings(P)$ which is a proper subset of the possible markings that a given PNP P may reach.

Definition 3. Executable PNP. *A PNP P is executable iff it exists a finite sequence of markings $\{M_0, ..., M_n\}$, such that M_0 is the initial marking, M_n is a goal marking (i.e. $M_n \in GoalMarkings(P)$) and $M_i \rightarrow M_{i+1}$, for each $i = 0, ..., n-1$.*

Definition 4. Correct execution of a PNP. *An executable PNP P can be correctly executed iff there exist a finite sequence of situations $\{\tau_0, ..., \tau_{n-1}\}$ and a finite sequence of markings $\{M_0, ..., M_n\}$, such that M_0 is the initial marking, M_n is a goal marking (i.e. $M_n \in GoalMarkings(P)$) and $M_i \Rightarrow_{\tau_i} M_{i+1}$, for each $i = 0, ..., n-1$.*

4.1 PNP Execution Algorithm

Algorithm 1, presented above, correctly executes a PNP. The algorithm computes a sequence of transitions $\{M_0, ..., M_n\}$ that evolve the system from the initial marking (i.e. M_0) to a goal marking (i.e. $M_n \in GoalMarkings$), according with the sequence of situations $\{\tau_0, ..., \tau_{n-1}\}$ occurring in the environment.

At each step it checks if each transition $t \in T$ can be fired (Algorithm 1, line 4). This requires to verify that: i) the transition is enabled and ii) in the current situation $\tau_{current}$ the event condition $t.\phi$ (usually a propositional formula) of the transition t is true. The evaluation of a condition (Algorithm 1, line 4) is performed by activating a corresponding function that evaluates the property from the internal state of the robot.

Algorithm 1 PNP Execution Algorithm

```
1: $CurrentMarking = InitialMarking$
2: while $CurrentMarking \notin GoalMarkings$ do
3:   for all $t \in T$ do
4:     if $enabled(t) \wedge eval(t.\phi)$ then
5:       if $t.hasAction()$ then
6:         $handleAction(t)$
7:       end if
8:       $CurrentMarking = fire(t)$
9:     end if
10:  end for
11: end while
```

Algorithm 2 The Action Handler

```
procedure $handleAction(Transition\ t)$
1: $CurrentAction = t.getAction()$
2: if $t.isStart()$ then
3:   $CurrentAction.start()$
4: else if $t.isEnd()$ then
5:   $CurrentAction.end()$
6: else if $t.isInterrupt()$ then
7:   $CurrentAction.interrupt()$
8: end if
```

Recall that we rely on a heterogeneous hybrid architecture (Section 1). This implies that we represent knowledge both at an operational and deliberative level. In general, both representations of the knowledge are consistently maintained in a *world model* which summarizes the current state (e.g. current distance from the ball and reliability of the information).

In our framework, the knowledge at the deliberative level is used to evaluate the event conditions in the PNPs. Each time we have to evaluate a condition guarding an enabled transition we query the world model in order to interpret the propositions composing it. For example, if a condition to evaluate includes the proposition `haveBall`, we will query the world model for the distance to the ball being smaller than a given small value.

Thus, if the transition is executable and belongs to an action structure, the procedure $handleAction$ (Algorithm 2) takes care of appropriately activating or deactivating the related action. The details of how this is done depend on the actual implementation of the system.

Finally, the algorithm fires the firable transition t_i updating the marking accordingly to the *firing rule*.

The algorithm correctly executes a PNP if the sequence $\{\tau_0, ..., \tau_{n-1}\}$ allows for it, as shown by the following theorem.

Theorem 1. *If* $\{\tau_0, ..., \tau_{n-1}\}$ *is a finite sequence of situations such that a PNP can be correctly executed, then Algorithm 1 computes a sequence of transitions*

$\{M_0, ..., M_n\}$, such that M_0 is the initial marking, M_n is a goal marking, and $M_i \Rightarrow_{\tau_i} M_{i+1}$, for each $i = 0, ..., n-1$.

Proof. We want to prove that Algorithm 1 computes a sequence of transitions $\{M_0, ..., M_n\}$, such that M_0 is the initial marking, M_n is a goal marking, and $M_i \Rightarrow_{\tau_i} M_{i+1}$, for each $i = 0, ..., n-1$. Trivially the first marking M_0 is the initial marking (Algorithm 1, line 1). Furthermore, in order for the algorithm to halt, the final marking must be a goal marking (Algorithm 1, line 2). Thus, $M_n \in GoalMarkings$.

The transition from a marking M_i to a marking M_{i+1} is obtained firing (Algorithm 1, line 8) a transition t_i. A necessary condition for firing is that t_i is enabled (Algorithm 1, line 4).

If t_i is enabled this means that each input place p_i (i.e. $(p_i, t) \in F$) is marked with at least $w(p_i, t)$ tokens. Since we assume $0 \leq w(p_i, t) \leq 1$ this implies that $\forall p' \in P$, s.t. $(p', t) \in F$, then $M_i(p') > 0$.

When an enabled transition t fires according to the firing rule, $w(p_i, t)$ tokens are removed for each input place p_i and $w(t, p_o)$ are added to each output place p_o such that $(t, p_o) \in F$. Thus given the assumption that $0 \leq w(p_i, t) \leq 1$, we have $M_{i+1}(p') = M_i(p') - 1$ for each $p' \in P$, s.t. $(p', t) \in F$ and $M_{i+1}(p'') = 1$ for each $p'' \in P$, s.t. $(t, p'') \in F$. Thus, each transition performed by the algorithm is a *possible transition*.

Finally, the algorithm ensures *executable transitions* checking that $t.\phi$ is verified at the current situation τ before firing t (Algorithm 1, line 4). $\square$

5 Multi-Agent Plans

Describing multi-agent plans has been considered either as *plan sharing* (or *centralized planning*), where the objective is to distribute a global plan to agents executing them, or as *plan merging*, where individual plans are merged into a global plan (see [Dur99] for details). In our work we followed the *centralized planning* approach that has been easily implemented in our formalism as described in this section.

A Multi-Agent PNP, for agents $\{1, \ldots, n\}$, can be defined as the union of n single agent PNPs enriched with synchronization constraints between actions of different robots.

When writing a Multi-Agent plan, the syntax is not much different from the single robot case, except that actions are labeled with a unique id for the robot. Given n single agent plans appropriately labeled $\{PNP_i =< P_i, T_i, F_i >\}$, the simplest way to define a Multi-Agent plan is:

$$M\_PNP =< M\_P, M\_T, M\_F >$$

where:

- $M\_P = \bigcup_{i=1}^{n} P_i$
- $M\_T = \bigcup_{i=1}^{n} T_i$

- $M\_F = \bigcup_{i=1}^{n} F_i$

Such a Multi-Agent plan consists simply of n independent plans. When dealing with Multi-Agent systems, the main issue is how to represent the interactions among actions performed by different agents (i.e. among plans). The Multi-Agent plan, as previously defined, fails to capture such interactions and may result in the execution of conflicting actions. In particular, we want to be able to order actions across plans so that overall consistency is maintained and conflicting situations are avoided.

For example, consider two robots cooperating in a foraging task (see the Multi-Robot testbed in Section 6).They must at first help each other to allow one of the robots to grab the object, then this robot can transport the object to a collect point, while the support of the second robot is not necessary anymore, and it should move away for not interfering with the first robot. Action synchronization is thus needed first for coordinating the grab action, then to communicate that the supporting robot is out of the way.

In our approach, we use action synchronization to avoid unsafe interactions. We will assume that the agents will be able to communicate through a reliable channel and thus to send and receive synchronization messages. The synchronization operator is defined as follows

$$SYNC =< \{p_c\}, \{t_{c_i}, t_{c_j}\}, \{(t_{c_i}, p_c), (p_c, t_{c_j})\} >$$

where:

- $i, j \in \{1, \ldots, n\}$
- $i \neq j$

The synchronization operator is used to add temporal constraints in the execution of the actions in a multi-agent plan. For example, Figure 11 describes a constraint between the execution of two actions performed by agents R1 and R2. The figure shows the representation of a constraint indicating that the action of agent R1 must start after the termination of the action of agent R2. While in Figure 12 we show how to represent the simultaneous execution of two actions by two agents R1 and R2. Note that network delay may affect exact simultaneous starting of the two actions; however, the formalism ensures that the two actions will be generally executed at the same time by the two robots.

Using the synchronization operator, we can thus write Multi-Agent PNPs in which all the conflicts in the actions are solved. Moreover, given a Multi-Agent PNPs, we can automatically produce the single-agent plans by isolating the portion of the plans relative to each robot and replacing synchronization operators with communication actions.

In particular, the synchronized Single-Agent plan $S\_PNP_i =< S\_P_i$, S_T_i, $S\_F_i >$, will be the minimal net such that:

$$P_i \subset S\_P_i \ \wedge T_i \subset S\_T_i \ \wedge F_i \subset S\_F_i \qquad (1)$$

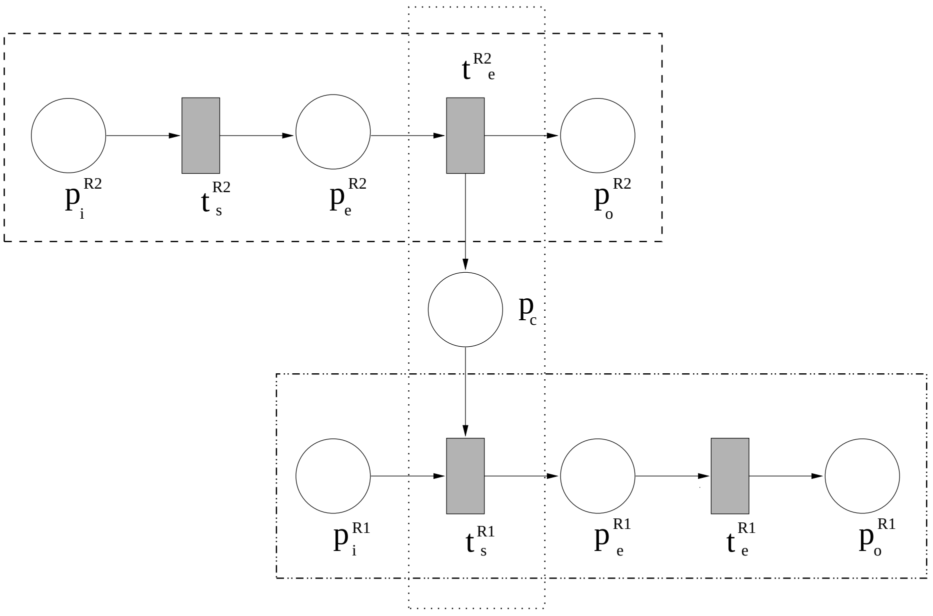

Fig. 11. Two actions of different agents which must be executed in sequence.

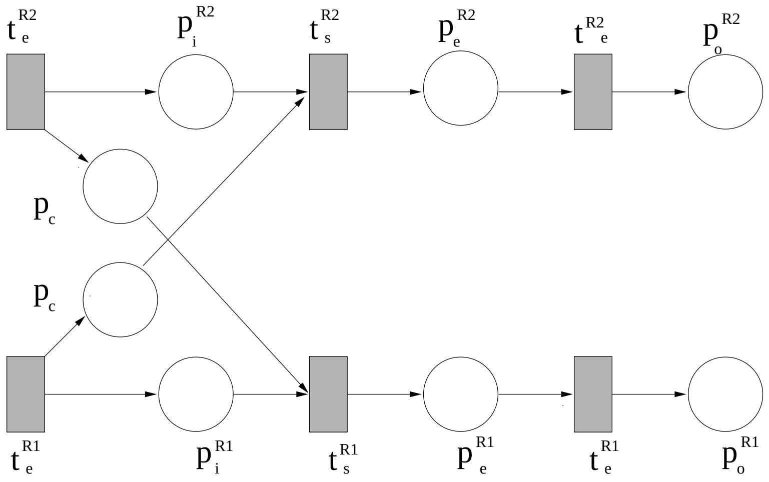

Fig. 12. Two actions of two different robots which must start at the same time.

$$\begin{gathered}\forall t_i \in T_i \; \forall p \in M\_P \; (\; \exists t_j \in T_j \wedge i \neq j \wedge (t_i, p) \in M\_F \wedge (p, t_j) \in M\_F) \Rightarrow \\ p \in S\_P_i \wedge \\ t_{send(t_i,t_j)} \in S\_T_i \wedge \\ (p, t_{send(t_i,t_j)}) \in S\_F_i \;\wedge\; (t_i, p) \in S\_F_i \end{gathered} \tag{2}$$

$$\begin{gathered}\forall t_i \in T_i \; p \in M\_P \; (\; \exists t_j \in T_j \wedge i \neq j \wedge (t_j, p) \in M\_F \wedge (p, t_i) \in M\_F) \Rightarrow \\ p \in S\_P_i \wedge \\ t_{rec(t_i,t_j)} \in S\_T_i \wedge \\ (t_{rec(t_i,t_j)}, p) \in S\_F_i \;\wedge\; (p, t_i) \in S\_F_i \end{gathered} \tag{3}$$

Condition 1 states that the synchronized plan must include the original single agent one. Condition 2 and Condition 3 state respectively that action communication primitives and message reception events must be added to the plan.

When the transition $t_{send(t_i,t_j)}$ is enabled the agent i will fire it and thus perform an instantaneous action which sends a synchronization message relative to the transitions t_i and t_j to agent j. When agent j receives such a message will store it and fire $t_{rec(t_i,t_j)}$ when enabled.

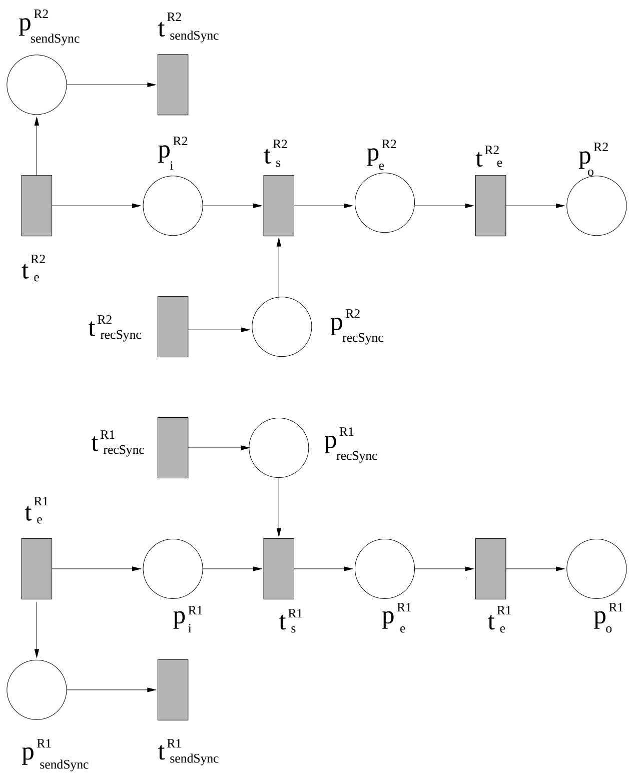

Fig. 13. The single agent plans extracted from the Multi-Agent one in Fig. 12

An example of such a process is shown in Figure 13. Here the Multi-Agent PNP of Figure 12 is divided in two PNPs for the two agents, and the synchronization operator is replaced by *Send* and *Receive* actions. The synchronized single agent plans are then executed as shown in Section 4. The communication primitives will guarantee the consistency of the distributed Multi-Agent plan.

6 Implemented Systems

The proposed framewok has been implemented and used to control different robotic systems in different domains.

A plan executor for our formalism has been implemented with a set of tools for designing and debugging plans. Plans are executed also according to the events occurring in the environment and to the state of the robot, which represents the agent's knowledge about the environment. During the execution of a PNP, the robot makes use of a set of functions that can access the internal state of the robot and return truth values about relevant properties for the execution of the plan, and information about the state of knowledge about such properties. For example, the robot may use a function returning whether the position of the ball is known (i.e. the ball is visible to the robot's sensors), and another function returning an evaluation of the fact that the ball is close enough to be kicked.

Plans can be generated by an off-line planner (currently without concurrency) or edited by hand. In the latter case we use an available open-source graphical tool, Jarp[3], which can generate an appropriate standard XML format (PNML). If transitions and places are correctly labeled to meet the specification of the Petri Net Plan the PNML code is parsed to produce executable representation on the robot (PET files).

Moreover, Jarp has been extended in order to debug plans on-line. During plan execution, the robot can produce (or stream through a TCP/IP connection) a log containing the information regarding the deliberative process. This log can be parsed by our tool to view the evolution of the Petri Net Plans, allowing for easily identifying loops or wrong behaviors, and providing a quick and user friendly plan debugging interface.

The Petri Net Plans are used for designing the behaviors of the Robocup 4Legged team S.P.Q.R. Legged[4] since 2004 [IN04]. In this league, two teams of four autonomous Sony Aibo, play a soccer match on a rectangular field with a set of landmarks in known positions. The approach has been successful in modeling behaviors in such a highly dynamic and noisy environment. The robots were able to handle reactively rapid state changes while demonstrating a proactive behavior allowing for a good performance in the competitions.

On the other hand, Petri Net Plans have been employed to design behaviors for quasi-static environments, where the focus is on information gathering. This is the

[3] http://jarp.sourceforge.net/
[4] http://spqr.dis.uniroma1.it/

case of the Real Robot Rescue competitions where the goal is to explore and seek for victims in an unstructured environment (i.e a disaster scenario like a building after an earthquake).

The S.P.Q.R. Real Rescue team[5] adopts the Petri Net Plans since 2005 to control their rescue robots. The use of Petri Net Plans to model urban search and rescue scenarios has been one of the topics of the practical sessions at the Rescue Robotics Camp[6].

Finally, we used the Petri Net Plans to design a set of experiments for a task assignment technique based on token passing[7] [FINZ06]. Our application scenario was formed by a set of robots that need to perform a synchronized operation on a set of similar objects scattered in the environment. In order to achieve such a complex foraging task it is necessary to be able to synchronize actions across plans as shown in Section 5. In particular, we implemented the communication through TCP/IP triggering events based on reception of appropriate sync messages.

7 Discussion

The experience in using Petri Net Plans for programming our robots has been very effective, providing for many advantages over other techniques, as well as some difficulties that we have dealt with. In this section we want to analyze the main advantages and possible drawbacks of this formalism.

The main advantage of the Petri Net Plan framework is the clear definition of the modeling language and of its semantics in terms of Petri nets. We have chosen to adopt the Extended Petri nets because it is the simplest model necessary to specify the constructs we needed to model. Moreover, if the definite operator is not used, PNPs are a subset of the basic Petri nets. Using such a model, rather than one of its many extensions, guarantees us the possibility to use standard tools to evaluate properties of the nets such us liveness and reachability of the goal states.

The gain in using Petri nets is that we have a formal method to distinguish action implementation and specification. Moreover, the graphical representation of Petri nets allows for an easy understanding and debugging of the plans which speeds up the development process. High expressiveness of PNPs thus allows for effectively capturing and dealing with most of the situations encountered when designing autonomous robots.

On the other hand, such high expressiveness is also a limitation when designer is interested in using plan generation techniques. Therefore, it is necessary for the user to manually write the plans for the agents or enhance automatically generated plans for handling concurrency.

Although we provide an operational semantics for our plans, in order to have a clear specification of the behavior of the robots during execution, it may still be

[5] http://sied.dis.uniroma1.it/

[6] http://sied.dis.uniroma1.it/camp/

[7] A video of the experiment is available at:
http://www.dis.uniroma1.it/∼farinell/video/CoopForaging-commentary.wmv

very difficult to debug plans when their size grows and the dependency across them becomes very complex. This is especially true for Multi-Agent plans. At the moment we rely on the user to design correct plans and to solve related problems.

The problem of plan correctness is a common problem in behavior design and has been addressed in the literature in different ways. In particular, we can roughly categorize related approaches in three main classes.

1. Hand-written behaviors directly coded in robot program. In this case there is no explicit representation of actions and plans. It is thus very difficult to design, write and debug plans.
2. Hand-written behaviors using behavior oriented languages (e.g. Xabsl [LBBJ04] and Colbert [Kon97]). These languages consist of behavioral routines, but, although a framework for designing plans is defined, there is no formal specification and thus it is not possible to verify properties of these programs/behaviors.
3. Logic-based programming (e.g. Golog [Rei01]). These are declarative languages with reasoning abilities. In particular, in these frameworks behaviors are specified in a high level programming language based on some formal system (e.g. Situation Calculus). Such programs allow not to specify all the details of the program which are computed by a reasoning system. The main drawback of such approaches is that they are computationally very expensive and are inadequate to control very complex real time systems.

Our approach lies between the second and the third category. On the one hand, as for other behavior oriented languages, we provide for an efficient framework for designing, writing, executing, and debugging plans, which explicitly represents actions and plans. On the other hand, as in logic based programming, we provide a formal specification of our plans which allows for implementing reasoning and verification procedures. In fact, we are working on integrating formal action specification in the PNP in order to verify properties of plans such as correctness and termination.

Our formalism differs from Golog language also in the representation of the properties in the environment. In PNP it is possible to model only the knowledge (or the absence of knowledge) of the agent about the environment, while it is not possible to model what is actually true in the environment. In other words, the agent acts only on the basis of what it knows about the environment: knowledge is acquired either by direct perception (i.e., analysis of sensor data) or by the assumption that the effects of an action hold when this action has been correctly executed. To this end we make explicit use of sensing actions (or knowledge producing actions), as in [SL93, DGINR97].

8 Conclusions and Future work

In this paper we have presented a modeling language to design deliberative layers of agents/robots based on a *heterogeneous hybrid* architecture which inhabit a dynamic, partially observable and unpredictable environment.

As already mentioned, this modelling tool has been deeply tested and implemented in different scenarios. We can thus enforce the adequacy of the approach based on experimental evidence. In particular, we have seen that the high flexibility of the language, the modular development and the easy to use tools help the user in the design task and our many students that used the tool to write, execute and debug complex behaviors quickly and with a small effort.

The most critical problem we faced when specifying PNPs was to define a semantics in order for the user to have a clear specification of the behavior of the robots during execution. This problem was solved by defining an operational semantics and proving the correctness of its execution. Nevertheless, it may still be the case that conflicts arise when executing parallel plans.

As future work, we are planning to implement verification and plan assistant tools in order to guarantee the safeness of plans. In order to do this, we will need to provide a formal description of actions using some action specification language. In particular, we are studying a more formal relationship between the presented modelling language and logic-based formalisms for reasoning about actions (such as, ConGolog [DLL00]). In this direction, we are currently investigating a possible extension of a formalism for reasoning about actions based on Description Logic [BCM+03], that has been previously used for generating high-level programs for mobile robots [INR00, ILNR04, INR04].

References

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[DGINR97] Giuseppe De Giacomo, Luca Iocchi, Daniele Nardi, and Riccardo Rosati. Planning with sensing for a mobile robot. In *Proc. of 4th European Conference on Planning (ECP'97)*, 1997.

[DLL00] G. DeGiacomo, Y. Lesperance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[Dur99] Edmund H. Durfee. Distributed problem solving and planning. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 121–164. MIT Press, 1999.

[FINZ06] A. Farinelli, L. Iocchi, D. Nardi, and V. A. Ziparo. Assignment of dynamically perceived tasks by token passing in multi-robot systems. *Proceedings of the IEEE, Special issue on Multi-Robot Systems*, 2006. To appear.

[Fir89] R. James Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale, 1989.

[GL86] M. P. Georgeff and A. L. Lansky. Procedural knowledge. In *Proceedings of the IEEE Special Issue on Knowledge Representation*, volume 74, pages 1383–1398, 1986.

[ILNR04] L. Iocchi, T. Lukasiewicz, D. Nardi, and R. Rosati. Reasoning about actions with sensing under qualitative and probabilistic uncertainty. In *Proc. of 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 818–822, Spain, 2004.

[IN04] Luca Iocchi and Daniele Nardi. SPQR-Legged Team 2004. In *RoboCup 2004: Robot Soccer World Cup VIII*. Springer-Verlag, 2004.

[INR00] L. Iocchi, D. Nardi, and R. Rosati. Planning with sensing, concurrency, and exogenous events: logical framework and implementation. In *Proc. of KR'2000*, 2000.

[INR04] L. Iocchi, D. Nardi, and R. Rosati. Strong cyclic planning with incomplete information and sensing. In *Proc. of 4th Int. Workshop on Planning and Scheduling for Space*, Darmstadt, Germany, 2004.

[Kon97] K. Konolige. COLBERT: A language for reactive control in sapphira. *Lecture Notes in Computer Science*, 1303:31–50, 1997.

[LBBJ04] Martin Ltzsch, Joscha Bach, Hans-Dieter Burkhard, and Matthias Jngel. Designing agent behavior with the extensible agent behavior specification language XABSL. In Daniel Polani, Brett Browning, and Andrea Bonarini, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*, pages 114–124, Padova, Italy, 2004. Springer.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[PDPW] Barney Pell, Gregory A. Dorais, Christian Plaunt, and Richard Washington. The remote agent executive: Capabilities to support integrated robotic agents.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Rei01] R. Reiter. *Knowledge in action: Logical foundations for describing and implementing dynamical systems*. MIT Press, 2001.

[SA98] Reid Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings Conference on Intelligent Robotics and Systems*, October 1998.

[SL93] Richard Scherl and Hector J. Levesque. The frame problem and knowledge producing actions. In *Proc. of AAAI-93*, pages 689–695, 1993.