Petri Net Plans

A Formal Model for Representation and Execution of Multi-Robot Plans

V. A. Ziparo[1], L. Iocchi[1], D. Nardi[1], P.F. Palamara[1], H. Costelha[2] *

[1]Dipartimento di Informatica e Sistemistica
Via Ariosto 25, I-00185
Rome, Italy
{ziparo,iocchi,nardi}@dis.uniroma1.it

[2]Institute for Systems and Robotics, IST, Lisboa
Polytech. Inst. of Leiria, ESTG, Leiria
Portugal
hcostelha@isr.ist.utl.pt

ABSTRACT

The aim of this paper is to describe a novel representation framework for high level robot and multi-robot programming, called Petri Net Plans (PNP), that allows for representing all the action features that are needed for describing complex plans in dynamic environments. We provide a sound and complete execution algorithm for PNPs based on the semantics of Petri nets. Moreover, we show that multi-robot PNPs allow for a sound and complete distributed execution algorithm, given that a reliable communication channel is provided. PNPs have been used for describing effective plans for actual robotic agents which inhabit dynamic, partially observable and unpredictable environments, and experimented in different application scenarios.

1. INTRODUCTION

High level programming of mobile robots is very important for developing complex and reliable robotic applications. High level programming is usually performed by defining *plans*, i.e. program structures describing action execution control. To develop complex applications, plans should represent many features such as sensing, loops, concurrency, non-instantaneous actions, action failures, and action synchronization in a multi-agent context. We can roughly classify high-level robot programming methods as follows.

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 [8] and Colbert [7]). These languages consist of behavioral routines, but, although a framework for designing plans is defined, there is no formal specification of the language and thus it is not possible to verify properties of these programs/behaviors.
3. Logic-based programming (e.g., ConGolog [4]). These are declarative languages with associated reasoning capabilities. In these frameworks behaviors are specified in a high level programming language based on a formal system. The main drawback of such approaches is that they are computationally very expensive and inadequate to control very complex real time systems.

Our approach lies between the second and the third categories. On the one hand, as for other behavior oriented languages, we provide for an efficient framework for designing, writing, executing, and debugging plans. On the other hand, as in logic based programming, we clearly distinguish action specification and implementation and we provide a formal specification of our plans which allows for implementing reasoning and verification procedures.

The aim of this paper is thus to describe a representation framework for high level robot and multi-robot programming, called *Petri Net Plans* (PNP)[13], that allows for representing all the action features that are needed for describing complex plans in dynamic environments. PNPs are based on Petri nets [9], a graphical modeling language for dynamic systems, and used for describing effective plans for actual robotic agents which operate dynamic, partially observable and unpredictable environments.

Petri nets (PNs) have been already used to model robotic behaviors, as for example [2] and [12]. The former provides an interesting formal approach for modeling single-robot tasks, while the latter uses PNs to model a multi-robot coordination algorithm, based on an auction mechanism, to perform environment exploration. However, both approaches have only been evaluated through simulation. Our approach allows the modeling of generic multi-robot tasks, and has been evaluated on real robots. Moreover our approach differs from these works since PNPs are particular Petri nets that embed some of the features of the logics of actions, while keeping the characteristics of a dynamic computational model. More specifically, if we consider an action theory A and a PNP formed as a composition of actions in A that respects the semantics of the action theory, then any execution of the PNP (i.e., a sequence of markings corresponding to a sequence of states) from an initial marking (state) to a goal marking (state) is a solution of a planning problem defined by the given action theory A, initial state and goal state. This property allows for implementing interesting forms of execution monitoring and to state correctness of PNP execution according to an action theory.

In the remainder of the paper, after introducing basic notation in the next section, we define in Section 3 the syntax for single-robot PNPs 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 re-

*The work of this author was supported by the Portuguese FCT through grant SFRH/BD/12707/2003.

Cite as: PNP: A Formal Model for Representation and Execution of Multi-Robot Plans, V. A. Ziparo, L. Iocchi, D. Nardi, P.F. Palamara, H. Costelha, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. XXX-XXX.
Copyright © 2008, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

covery; 2) sensing non-instantaneous actions, which allow for dynamically sensing properties at execution time and thus for knowledge acquisition [11, 3]. We 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 prove that, if a correct execution is possible, then the algorithm will achieve it. We then define multi-robot plans (Section 4) as collections of single-robot PNPs, coordinated through synchronization actions which are used to spread information and synchronize actions of different robots. We show that multi-robot PNPs can be decomposed into a set of single-agent PNPs, whose distributed execution is equivalent to the centralized execution of the original multi-robot PNP. Experimental tests are finally described in Section 5.

2. PETRI NETS

Petri nets [9] 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.

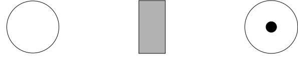


Figure 1: (a) A place. (b) A Transition. (c) A Place with one token.

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

$$PN = \langle P, T, F, W, M_0 \rangle$$

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 $(t, p_o) \in F$.

3. SINGLE-ROBOT PNPS

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.

3.1 Syntax

A *Petri Net Plan*

$$\langle P, T, F, W, M_0, G \rangle$$

is a Petri net $\langle P, T, F, W, M_0 \rangle$ augmented with a set of goal markings G such that:

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.

5. G is the set of desired markings for the agent and is a proper subset of the possible markings that the PNP may reach.

In the following we will focus on the structure of a PNP (i.e. considering only the terms $\langle P, T, F \rangle$). 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). The following description of single-robot PNPs is provided in terms of the graphical representation of Petri nets (see [13] for a detailed description).

3.1.1 Elementary structures

Elementary PNPs are defined as follows:

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

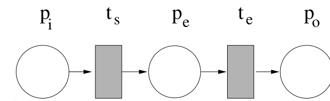


Figure 2: An ordinary non-instantaneous action.

2. **ordinary-action**, depicted in Figure 2, is a PNP defined by 3 places and 2 transitions where:

   - p_i, p_e and p_o are the initial, execution and terminating place, respectively.
   - t_s and t_e are the transitions starting and terminating the action, respectively.

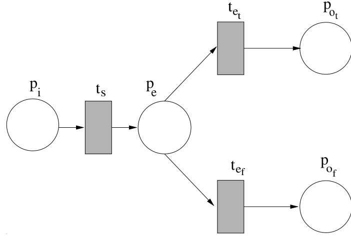


Figure 3: A non-instantaneous sensing action.

3. **sensing-action**, depicted in Figure 3, is a PNP defined by places and transitions 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.

3.1.2 Operators

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

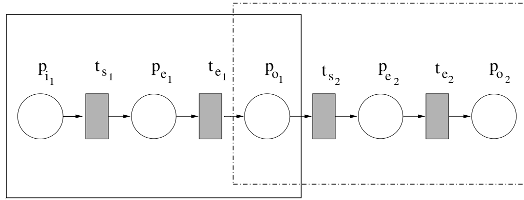


Figure 4: Sequence of two PNPs.

The *sequence* operator allows to sequence in time two PNPs. This operator allows for merging two PNPs by 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.

The *conditional* operator, depicted in Figure 5, allows for describing conditional structures that are implemented though sensing actions. Given a sensing action α, three PNPs Γ_1, Γ_2, Γ_3, the conditional structure is obtained by sequencing Γ_1 with α, and by sequencing each outcomes of α with either Γ_2 or Γ_3. The structure specifies that after executing Γ_1, depending on the outcome of α, Γ_2 or Γ_3 will be executed.

The *loop* operator, depicted in Figure 6, is used to obtain indefinite iterations while a sensed condition remains true. Its implementation is similar to a conditional structure, except that one sensing outcome is sequenced with Γ_1 forming a loop. The result is a net executing Γ_1 until the sensed property becomes true.

Concurrent execution of actions is defined by the *fork* and *join* operators. The fork operator, Figure 8(a), is obtained by generating two tokens (representing two parallel threads of execution) from a single token (representing a single thread of execution) through a control transition. In a similar way we can define the join operator, depicted in Figure 8(b), which produces one thread of execution (i.e. token), from two distinct ones.

Finally, we introduce the *interrupt* operator, depicted in Figure 7, which is a very powerfull tool for handling action failures. In fact,

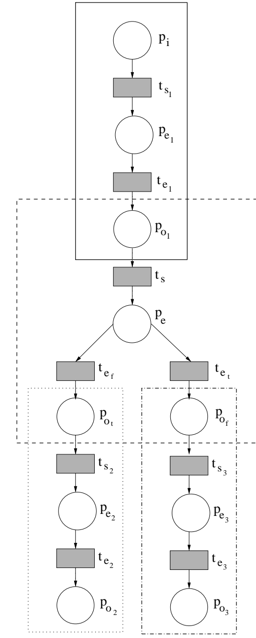


Figure 5: Conditional structure.

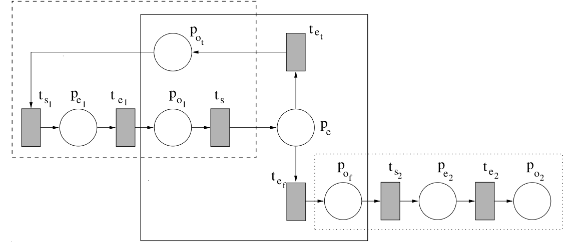


Figure 6: An *indefinite iteration* which executes the PNP Γ_1 while the sensed property is true.

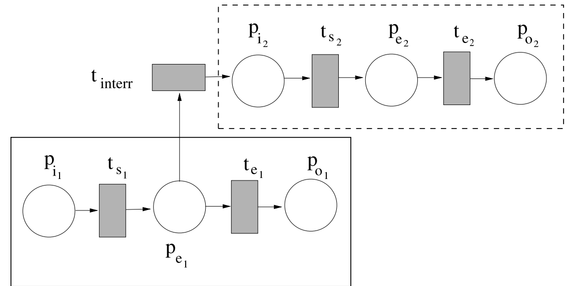
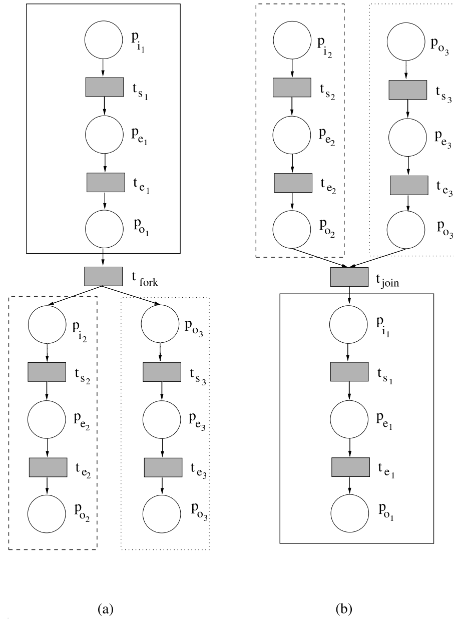


Figure 7: Interrupt structure where possibly Γ_1 is interrupted and then Γ_2 executed.

it can interrupt actions upon failure events and activate recovery procedures.

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 the condition of ending transitions to $False$



(a) (b)

Figure 8: (a) The fork structure. (b) The join structure.

to model non-terminating actions (for example, support actions run in parallel with another main action).

3.2 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.

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 by querying a knowledge base KB and thus determine when and how it is possible to perform such transitions.

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.

Definition 3.1 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 3.2 Executable transition in a PNP. *Given two markings M_i, M_{i+1} and a K_i at time i, a transition from M_i to M_{i+1} 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 $t.\phi$) holds in K_i (i.e. $K_i \models \phi$).*

An executable transition from M_i to M_{i+1} is denoted by $M_i \Rightarrow M_{i+1}$.

Definition 3.3 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. $M_n \in G$) and $M_i \rightarrow M_{i+1}$, for each $i = 0, ..., n-1$.*

Definition 3.4 Correct execution of a PNP. *An executable PNP P can be correctly executed iff there exist 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. $M_n \in G$) and $M_i \Rightarrow M_{i+1}$, for each $i = 0, ..., n-1$.*

3.2.1 PNP Execution Algorithm

Algorithm 3.1 PNP Execution Algorithm

Domains:

$\mathcal{A} = \{a_1, \ldots, a_k\}$ **:** Set of Implemented actions
Φ **:** Set of terms and formulas about the environment
$TrType = \{start,\ end,\ interrupt,\ standard\}$

Structures:

$Transition : \langle a \in \mathcal{A},\ \phi \in \Phi,\ t \in TrType \rangle$
$Action : \langle start(),\ end(),\ interrupt() \rangle$

Global Variables:

$KnowledgeBase : KB$

procedure $execute(PNP\ \langle P, T, F, W, M_0, G \rangle)$

1: $CurrentMarking = M_0$
2: **while** $CurrentMarking \notin G$ **do**
3: **for all** $t \in T$ **do**
4: **if** $enabled(t) \wedge KB \models t.\phi$ **then**
5: $handleTransition(t)$
6: $CurrentMarking = fire(t)$

procedure $handleTransition(t)$

if $t.t = start$ **then**
$t.a.start()$
else if $t.t = end$ **then**
$t.a.end()$
else if $t.t = interrupt$ **then**
$t.a.interrupt()$

In the following, we present an algorithm which correctly executes a PNP. Algorithm 3.1 assumes the availability of a set of implemented actions $\mathcal{A} = \{a_1, \ldots, a_k\}$. Each action considered here is an abstraction for the implementation of a specific behavior that the robots can execute: we assume the action can be accessed by the three functions *start*, *end* and *interrupt*, that, respectively, start, terminate and suspend the execution of such a behavior. We also assume that actual behavior execution will be performed in a separate thread with respect to the execution of Algorithm 3.1[1]. This means that after an action is started, it will remain active until either *end* or *interrupt* will be invoked.

Moreover, since we can not assume that the agent has complete knowledge about all the properties of the environment at each point in time, the evolution of the plan must be controlled according to the robot actual knowledge about the environment (i.e., according to its epistemic state of knowledge). Therefore, we assume that the robot maintains a knowledge base KB containing information

[1]Obviously, this can be easily extended to non-threaded cases.

about the environment. This knowledge base can be implemented in any form with any formalism: for example, on heterogeneous cognitive robots normally epistemic knowledge is represented both at an operational level (as data structures) and at a deliberative level (as predicates). Pairwise, queries over the environment Φ can be represented as terms or formulas in any formalism consistent with the knowledge base. For the purposes of our plan execution method, we only require that the agent is able to evaluate queries over the current model of the world, i.e., to calculate $KB \models t.\phi$.

The procedure $execute$ takes as input a PNP $\langle P, T, F, W, M_0, G\rangle$ and evolves it producing the control commands for the basic behaviors (which are associated to the firing of transitions). This process generates a sequence of transitions $\{M_0, ..., M_n\}$ that evolve the system from the initial marking M_0 to a goal marking $M_n \in G$.

In particular, at each step, Algorithm 3.1 checks (line 4) if each transition $t \in T$ is enabled ($enabled(t)$) and if the related event occurs. In our setting, an event occurs if the formula ϕ guarding t is satisfied given the current knowledge KB (i.e. $KB \models t.\phi$). If these two conditions are satisfied the transition t is fired (line 6) and the relative procedures for action control are handled within the sub-procedure $handleTransition$ (line 5) that takes care of appropriately activating, interrupting or deactivating the related action. The details of how this is done depend on the actual implementation of the system.

The algorithm correctly executes a PNP as shown by the following theorem.

Theorem 3.1 *[13] If a PNP can be correctly executed, then Algorithm 3.1 computes a sequence of transitions $\{M_0, ..., M_n\}$, such that M_0 is the initial marking, M_n is a goal marking, and $M_i \Rightarrow M_{i+1}$, for each $i = 0, \ldots, n-1$.*

4. MULTI-ROBOT PNPS

The design of multi-robot 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 multiagent plan (see [5] for details). In our work we followed the *centralized planning* approach that has been easily implemented in our formalism as described in this section. In particular, we show how to represent a multi-robot PNP which can be produced in a centralized manner and we provide a distributed execution model for it by implementing the *centralized planning for distributed plans* approach [5]. The distributed execution model allows to execute a set of single-robot PNPs, derived from the multi-robot PNP, without the need of a central coordinator agent. The correctness of the distributed execution with respect to the multi-robot PNP is enforced using the communication primitives *send(id)*, *receive(id)* and *sync(id,id')*, where *id* and *id'* are unique identifiers for the state of execution of single-robot plans, as we will show in the following. The primitives are modeled as single-robot ordinary non-instantaneous actions and represent communication acts.

4.1 Syntax

A multi-robot PNP, for agents $\{1, \ldots, n\}$, can be defined as the union of n single-robot PNPs enriched with synchronization constraints between actions of different robots. When writing a multi-robot 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-robot PNPs $\{\langle P_i, T_i, F_i\rangle\}$, appropriately labeled, the simplest way to define a multi-robot plan is:

$$M_PNP = \langle M_P, M_T, M_F\rangle$$

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-robot plan consists simply of n independent plans. When dealing with multi-robot systems, the main issue is how to represent the interactions among actions performed by different agents (i.e. among plans). The multi-robot 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.

In our approach, we model multi-robot plans as a collection of single-robot plans enriched with synchronization constraints to avoid unsafe interactions. In particular, we introduce new types of operators, assuming that robots can communicate through a reliable channel. In the following we describe a *hard synchronization* operator that synchronizes two plans in a given point in time and a *soft synchronization* operator which introduces a precedence relation among the actions of two plans. Another synchronization operator that allows for relating interrupts between the actions of two robots is shown within the example in Section 5.2.

4.1.1 Hard Synchronization

We define a *hard synchronization* operator among two robots s and r, depicted in Figure 9(a), and denote it $h_sync(s, r, id_s, id_r)$:

$$\langle \{p_{i1}, p_{i2}, p_c, p_{o1}, p_{o2}\}, \{t_s, t_e\}, \{(p_{i1}, t_s), (p_{i2}, t_s), (t_s, p_c), (p_c, t_e), (t_e, p_{o1}), (t_e, p_{o2})\}\rangle$$

The operator synchronizes in time two single-robot plans and allows for information share among them, through the communication of id_s and id_r which encode the state of execution for the plan of agent s and agent r, respectively. This operator is similar in structure to an ordinary non-instantaneous action, except that it does not belong to any agent and it is labeled with a unique pair (id_s, id_r).

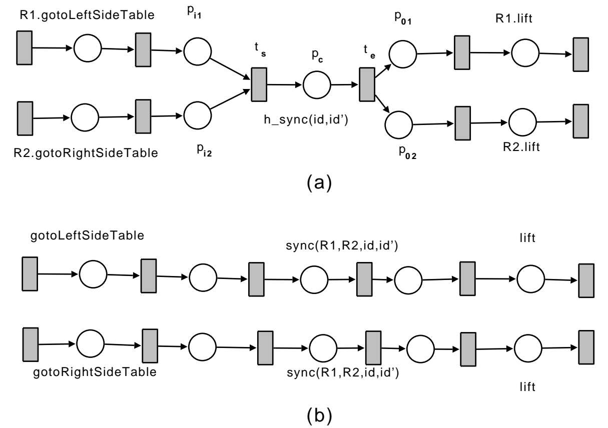


Figure 9: (a) A multi-robot PNP for hard synchronization. (b) The single-robot PNPs obtained from the multi-robot one.

For example, Figure 9(a) shows a multi-robot PNP for two robots which have to lift a table. The nodes for action structures and synchronization operators are grouped, for readability, by a common label. In this example $R1$ and $R2$ have to reach the two sides of a table and lift it simultaneously. The h_sync operator ensures that the robots will start to lift the table when both have reached it. In particular, the input transition t_s acts as a join waiting for both actions $R1.gotoLeftSideTable$ and $R1.gotoRightSideTable$ to terminate. The place p_c represents the state in which the communication, necessary for synchronization, is in progress. Finally, the

ending transition of t_e acts like a fork enabling the performance of the lift actions.

We now provide the formal definition of the hard synchronization operator. Consider, without loss of generality, a sequence of actions, $R1.act1$ and $R1.act2$, of robot R1, and a sequence of actions, $R2.act1$ and $R2.act2$, of robot R2. Assume that p_{or1} and p_{ir1} are the output and input place of $R1.act1$ and $R1.act2$, respectively. Moreover, assume that p_{or2} and p_{ir2} are the output and input place of $R2.act1$ and $R2.act2$, respectively. The multi-robot plan, which enforces $R1.act2$ and $R2.act2$ to start simultaneously, is the union of the four actions and the h_sync operator, with the constraint that:

$$p_{or2} = p_{i2} \wedge p_{ir2} = p_{o2} \wedge p_{or1} = p_{i1} \wedge p_{ir1} = p_{o1}$$

4.1.2 Soft Synchronization

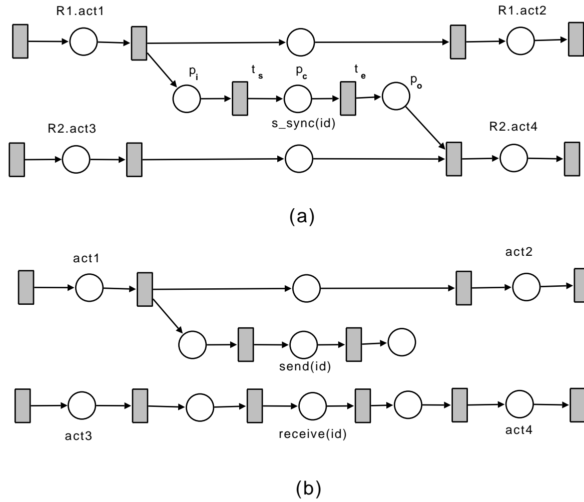


Figure 10: (a) A multi-robot PNP for soft synchronization. (b) The single-robot PNPs obtained from the multi-robot one.

We define a *soft synchronization* operator and denote it as s_sync:

$$\langle \{p_i, p_c, p_o\}, \{t_s, t_e\}, \{(p_i, t_s), (t_s, p_c), (p_c, t_e), (t_e, p_o)\}\rangle$$

This operator defines a precedence relation among two actions of two different robots. Figure 10(a) shows the representation of a soft synchronization enforcing that the action of agent R1 must start after the termination of the action of agent R2.

Consider, without loss of generality, the multi-robot plan composed by a sequence of actions, $R1.act1$ and $R1.act2$, of robot R1, and a sequence of actions, $R2.act3$ and $R2.act4$, of robot R2. We want to enforce $R1.act1$ to be executed before $R2.act4$. Assume that t_{er1} is the ending transition of $R1.act1$ and that t_{sr2} is the starting transition of $R2.act4$. The synchronized multi-robot plan is the original multi-robot plan merged with the s_sync operator and the new edges (p_o, t_{sr2}) and (t_{er1}, p_i). Figure 10(b) shows the single-robot plans obtained from this synchronized multi-robot plan.

4.2 Semantics

The semantics of a multi-robot PNP is the same of a single-robot PNP in the case of multibody planning [10], where a single centralized agent can dictate actions prescribed by the plan and query the knowledge base of each agent. Nevertheless, this approach is not desirable because it introduces a single point of failure in the system (i.e. the centralized agent).

We show that multi-robot PNPs allow for distributed execution. In particular, we provide an operational semantics for distributed execution. Roughly, given a multi-robot PNP, we can automatically produce a set of single-robot PNPs by isolating the portion of the plans relative to each robot and replacing synchronization operators with communication actions. Each single-robot plan can be locally executed by a robot without the need of a centralized coordinator, while correctness is maintained by communication actions.

A multi-robot plan can be decomposed into two single-robot plans (e.g. Figures 9(b) and 10(b)) by isolating actions labeled with the same agent id and by decomposing the sync operators into the two communication primitives.

For the hard synchronization operator $h_sync(id_1, id_2)$ the two communication primitives are $sync(id_1, id_2)$ and $sync(id_2, id_1)$, defined as follows:

$$sync(id_1, id_2) = \langle \{p_i^{id_1}, p_e^{id_1}, p_o^{id_1}\}, \{t_s^{id_1}, t_e^{id_1}\}, \{(p_i^{id_1}, t_s^{id_1}), (t_s^{id_1}, p_e^{id_1}), (p_e^{id_1}, t_e^{id_1}), (t_e^{id_1}, p_o^{id_1})\}\rangle$$

and

$$sync(id_2, id_1) = \langle \{p_i^{id_2}, p_e^{id_2}, p_o^{id_2}\}, \{t_s^{id_2}, t_e^{id_2}\}, \{(p_i^{id_2}, t_s^{id_2}), (t_s^{id_2}, p_e^{id_2}), (p_e^{id_2}, t_e^{id_2}), (t_e^{id_2}, p_o^{id_2})\}\rangle.$$

These two (single-robot) primitives, when performed jointly by robots s and r, establish a communication link between s and r, based on which a protocol for synchronization is started. In particular, each action, for example $sync(id_s, id_r)$ performed by s, at first sends the id_s encoding the state of execution its plan to r and, then, waits for id_r from r, which is acknowledged upon reception. Finally, it waits an acknowledgment of reception of id_s by r to terminate. Notice that the exchange of information is based on the ids which encode the state of execution of each single plan (e.g. which sensing branches are performed during execution). 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.

A soft synchronization operator $s_sync(id)$ can be decomposed into two communication primitives: a blocking $receive(id)$ and a non-blocking $send(id)$:

$$send(id) = \langle \{p_i^{s(id)}, p_e^{s(id)}, p_o^{s(id)}\}, \{t_s^{s(id)}, t_e^{s(id)}\}, \{(p_i^{s(id)}, t_s^{s(id)}), (t_s^{s(id)}, p_e^{s(id)}), (p_e^{s(id)}, t_e^{s(id)}), (t_e^{s(id)}, p_o^{s(id)})\}\rangle$$

and

$$receive(id) = \langle \{p_i^{r(id)}, p_e^{r(id)}, p_o^{r(id)}\}, \{t_s^{r(id)}, t_e^{r(id)}\}, \{(p_i^{r(id)}, t_s^{s(id)}), (t_s^{r(id)}, p_e^{r(id)}), (p_e^{r(id)}, t_e^{r(id)}), (t_e^{r(id)}, p_o^{r(id)})\}\rangle.$$

Consider the example in Figure 10 where we want to enforce that $act1$ precedes $act4$. Intuitively, a sender robot performs a $send(id)$ action to inform a receiver that he ended action $act1$. The action is performed on a separate thread because there is no need to wait for executing $act2$. Nevertheless, the receiver robot performs the $receive(id)$ on the main thread because it has to be sure that $act1$ has ended before performing $act4$

We now provide a formal characterization of the single-robot PNPs S_PNP_i obtained a the multi-robot one M_PNP. We denote with $\langle P_i \subseteq M_P, T_i \subseteq M_T, F_i \subseteq M_F\rangle$ the subset of M_PNP composed by the operators labeled with agent i. Recall that synchronization operators do not belong to any agent. Given a multi agent plan $M_PNP = \langle M_P, M_T, M_F\rangle$ the single-robot plan for agent i, $S_PNP_i = \langle S_P_i,\ S_T_i,\ S_F_i\rangle$, is the

minimal net such that:

$$P_i \subseteq S_P_i \ \wedge T_i \subseteq S_T_i \ \wedge F_i \subseteq S_F_i \quad (1)$$

$$\begin{aligned}&\forall t,t' \in T_i\ \forall p,p' \in M_P\\ &(p,p' \in h_sync(i,r,id_1,id_2) \wedge (t,p) \in M_F \wedge (p',t') \in M_F) \implies\\ &(\{p_i^{id_1},p_e^{id_1},p_o^{id_1}\} \subseteq S_P_i \wedge \{t_s^{id_1},t_e^{id_1}\} \subseteq S_T_i \wedge\\ &\{(t,p_i^{id_1}),(p_i^{id_1},t_s^{id_1}),(t_s^{id_1},p_e^{id_1}),(p_e^{id_1},t_e^{id_1}),\\ &(t_e^{id_1},p_o^{id_1}),(p_o^{id_1},t')\} \subseteq S_F_i) \quad (2)\end{aligned}$$

$$\begin{aligned}&\forall t,t' \in T_i\ \forall p,p' \in M_P\\ &(p,p' \in h_sync(S,i,id_1,id_2) \wedge (t,p) \in M_F \wedge (p',t') \in M_F) \implies\\ &(\{p_i^{id_2},p_e^{id_2},p_o^{id_2}\} \subseteq S_P_i \wedge \{t_s^{id_2},t_e^{id_2}\} \subseteq S_T_i \wedge\\ &\{(t,p_i^{id_2}),(p_i^{id_2},t_s^{id_2}),(t_s^{id_2},p_e^{id_2}),(p_e^{id_2},t_e^{id_2}),\\ &(t_e^{id_2},p_o^{id_2}),(p_o^{id_2},t')\} \subseteq S_F_i) \quad (3)\end{aligned}$$

$$\begin{aligned}&\forall t \in T_i\ \forall p \in M_P\ (\ p \in s_sync(id)) \implies\\ &(\{p_i^{s(id)},p_e^{s(id)},p_o^{s(id)}\} \subseteq S_P_i \wedge \{t_s^{s(id)},t_e^{s(id)}\} \subseteq S_T_i \wedge\\ &\{(t,p_i^{s(id)}),(p_i^{s(id)},t_s^{s(id)}),(t_s^{s(id)},p_e^{s(id)}),(p_e^{s(id)},t_e^{s(id)}),\\ &(t_e^{s(id)},p_o^{s(id)})\} \subseteq S_F_i) \quad (4)\end{aligned}$$

$$\begin{aligned}&\forall t \in T_i\ \forall p,p' \in M_P\\ &(p \in s_sync(id) \wedge (p,t) \in M_F \wedge (p',t) \in M_F) \implies\\ &(\{p_e^{r(id)},p_o^{r(id)}\} \subseteq S_P_i \wedge \{t_s^{r(id)},t_e^{r(id)}\} \subseteq S_T_i \wedge\\ &(p',t) \notin S_F \wedge \{(p',t_s^{r(id)}),(t_s^{r(id)},p_e^{s(id)}),(p_e^{r(id)},t_e^{r(id)}),\\ &(t_e^{r(id)},p_o^{r(id)}),(p_o^{r(id)},t)\} \subseteq S_F_i) \quad (5)\end{aligned}$$

Condition 1 states that the synchronized plan must include i's single-robot part of the plan, but does not take into account synchronization. On the one hand, Conditions 2 and 3 ensure that, respectively, the send and receive primitives are correctly substituted to each hard synchronization. On the other one hand, in a similar way, Conditions 4 and 5 enforce the correct interpretation of the soft synchronization. Notice that synchronization actions are also used for exchanging relevant information for cooperation (see the passing experiment in the next section for an example). These communications allow robots to maintain local KBs.

Examples of this process are shown in Figures 9 and 10. Here the multi-robot PNPs in the first part (Figures 9(a) and 10(a)) are divided in two PNPs for the two agents (Figures 9(b) and 10(b)), where the synchronization operators are replaced by *send*, *receive* and *sync* actions. The synchronized single-robot plans are then executed as shown in Section 3.2. The communication primitives will guarantee the consistency of the distributed multi-robot plan.

The following theorem (see [1] for the proof) ensures correctness of distributed execution of a multi-robot PNP.

Theorem 4.1 *The execution of a multi-robot PNP is equivalent to the distributed execution of the single-robot PNPs derived from it.*

5. EXPERIMENTAL TESTS

The proposed framework has been implemented and used to control different robotic systems in different domains. A plan executor for PNP formalism has been implemented with a set of tools for designing and debugging plans. Plans are executed reacting to the events occurring in the environment and to the state of the robot. 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.

Among the many applications, we describe here two experimental tests implemented with AIBO robots: a cooperative foraging test and a passing test. The objective of these tests is to highlight the features of the formalism in representing the multi-robot plans needed to accomplish them. Complete multi-robot plans, derived single robot plans and videos showing the execution of these tasks with the AIBO robots are available at [1].

5.1 Cooperative foraging

The cooperative foraging test we have considered is set by three robots that 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. Each robot can take one of two tasks: *collector*, that grabs the object (a ball), *supporter*, that supports the collector robot during the grabbing phase. Tasks are assigned to the robots by an external module [6] that dynamically decides which robot is in best condition to execute the task. While multi-robot PNPs are used here to specify the coordinated behavior of the two robots that execute the *collector* and *supporter* tasks.

Figure 11 shows the multi-robot plan. Hard synchronization is used to synchronize the robot after they reach the corresponding target positions. Then the collector robot waits for the supporter one to push the ball below his neck. After that collector robot grabs the ball and the supperter robot moves away. Finally, the collector robot brings the object in the target area. All these synchronization activities are implemented on the robots by pairs of communication actions.

5.2 Passing

The passing test has the objective of showing the ability of the multi-robot PNP formalism to address both action synchronization and dynamic task assignment. In contrast with the previous test, here there is no external module for dynamic task assignment, while this is accomplished by using the PNP structures described in the previous sections. Two interesting portions of the multi-robot plan are shown in Figure 12. In the first picture, the hard synchronization construct ensures that the robots synchronize their activities after they both see the ball. Within the synchronization message, the robots also exchange their knowledge about the position of the ball. Then role assignment is based on sensing actions and the one which is closer to the ball will go and grab the ball, while the other will prepare to receive the pass. Note that this assignment is dynamic and depends on the actual position of the ball. Another interesting feature of this plan is shown in the second picture, where we show how an interrupt in the execution of an action performed by one robot (i.e., the robot loses the ball when it is grabbing it) implies the interrupt of an action in the other robot (i.e., the other robot will interrupt the 'receive-pass' action). This is achieved through a h-sync and it allows the robots to restart the coordination activity (dashed lines in the figure): for example, if the second robot is now closer to the ball, it will start the grabbing phase.

6. CONCLUSIONS

In this paper we have presented a new formalism for high level programming of multi-robot systems that is able to represent *plans* with many important features such as sensing, loops, concurrency,

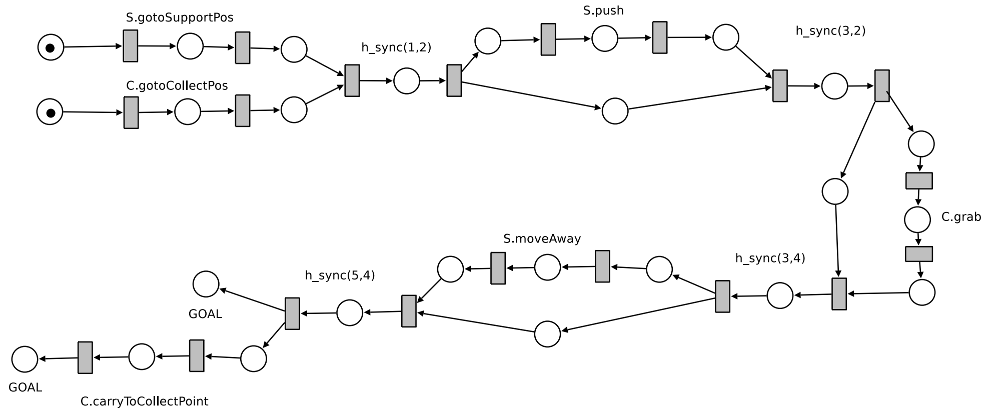


Figure 11: Multi-robot PNP for foraging test.

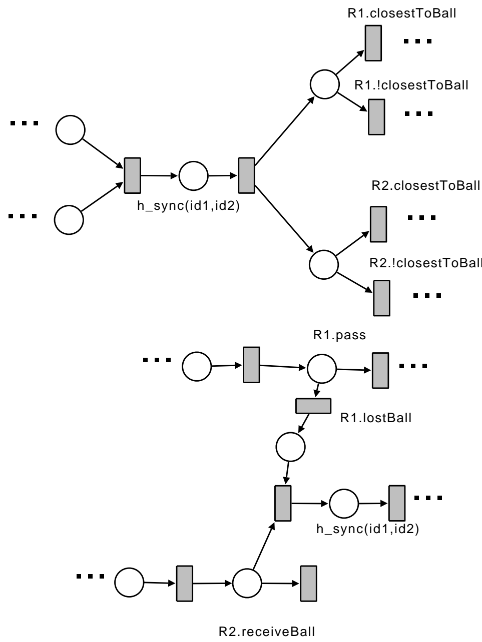


Figure 12: Portion of Multi-robot PNP for passing test.

non-instantaneous actions, action failures, and different types of action synchronization.

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. From one side, the high expressiveness of PNPs allows for effectively capturing and dealing with most of the situations encountered when designing autonomous robots and multi-robot systems. From the other side, we have a formal method to distinguish action implementation and specification and we can use standard tools to evaluate properties of the nets such us liveness and reachability of the goal states. Finally, the graphical representation of Petri nets allows for an easy understanding and debugging of the plans which speeds up the development process.

Such high expressiveness is also a limitation when designer is interested in using plan generation techniques. 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 difficult to write plans, especially Multi-Agent plans, for very complex tasks, like playing soccer.

Our future work will include two main streams: first, we want to study the possibility of extending plan generation techniques to automatically produce (at least partial) PNPs; second, we want to investigate learning techniques (e.g., genetic programming) for refining or generating plans by experiments and user training.

7. REFERENCES

[1] Extra Material Web Page. `www.dis.uniroma1.it/~ziparo/pnp_extras.html`.

[2] H. Costelha and P. Lima. Modelling, analysis and execution of robotic tasks using petri nets. *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 1449–1454, Oct. 29 2007-Nov. 2 2007.

[3] G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati. Planning with sensing for a mobile robot. In *Proc. of 4th European Conference on Planning (ECP'97)*, 1997.

[4] 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.

[5] E. 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.

[6] 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*, 94(7):1271–1288, 2006. ISSN:0018-9219.

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

[8] M. Lötzsch, J. Bach, H.-D. Burkhard, and M. Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. In D. Polani, B. Browning, and A. Bonarini, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*, pages 114–124, Padova, Italy, 2004. Springer.

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

[10] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. Second Edition.

[11] R. Scherl and H. J. Levesque. The frame problem and knowledge producing actions. pages 689–695, 1993.

[12] W. Sheng and Q. Yang. Peer-to-peer multi-robot coordination algorithms: petri net based analysis and design. *Advanced Intelligent Mechatronics. Proceedings, 2005 IEEE/ASME International Conference on*, pages 1407–1412, 24-28 July 2005.

[13] V. A. Ziparo and L. Iocchi. Petri net plans. In *Proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA)*, pages 267–290, Turku, Finland, 2006. Bericht 272, FBI-HH-B-272/06.