Petri Net Plans
A Framework for Cooperation and Coordination in Multi-Robot Systems

V.A. Ziparo · L. Iocchi · P. Lima · D. Nardi · P.F. Palamara

Abstract Programming behaviors for multi-robot systems is a challenging task and has a key-role in the development of effective systems in many application domains. In this paper, we present Petri Net Plans (PNPs), a Petri Net (PN) based language which allows an intuitive and effective robot and multi-robot behavior design. PNPs are very expressive and intuitive at the same time, allowing a small modeling effort for its users. As a central feature, PNPs allow formal analysis of plans based on PN standard tools. PNPs are suitable for modeling Multi-Robot Systems and the developed behaviors allow for distributed execution which maintains the properties of the modeled system. PNPs have been proved to be effective on several robotic platforms for many application domains. In this paper, we report two case studies which address coordination and cooperation issues.

Keywords Petri Nets · Multi-Robot Systems · Formal Models · Plan Representation and Execution

Mathematics Subject Classification (2000) 68T40 · 68T99 · 93C65 · 93C85

PACS 07.05.Mh · 07.07.Tw

CR Subject Classification 69K18 · 69K21 · 69K23

1 Introduction

The design of complex behaviors in dynamic, partially observable and unpredictable environments is a crucial task for the development of effective robotic applications. This is
particularly true when the task to accomplish requires coordination and cooperation among multiple robots, which have to act as a team, as teamwork can lead to consistent performance improvements.

In this work, we address the problem of establishing cooperation and coordination within a small team of robots that have to perform a complex task. Specifically, we assume that the task to be accomplished requires forms of cooperation (e.g., teamwork or task assignment) for robustness and efficiency, as well as complex action coordination, to avoid interference and enhance the capabilities of the system (e.g., joint actions and complex multi-robot plans).

Typically, multi-agent (or robot) plans can be achieved through:

- **Plan Design** (e.g., [LRJ06, Kon97]): based on a representation formalism, an expert designs by hand the behaviors, which allow for the accomplishment of a given task;
- **Plan Generation** (e.g., [GL05, DL91]): based on a description of the goals and the capabilities of a system, a planner generates a solution, whose execution achieves the task.

The former approach can be used to write very expressive plans which are limited solely by the expressiveness of the representational formalism used and by the capabilities of the designer. Nevertheless, it can be very hard to deal with such plans when they become large and complex in realistic applications. The latter approach is clearly more desirable, because it automates a task that requires considerable effort of specialized operators and is prone to errors. Nevertheless, the complexity of tasks for multi-robot systems, and in general for the physical world, limits the possibility of applying such approaches. Indeed, either they are not enough expressive to represent all the features of interest or they are too complex to compute solutions for realistic applications.

A different approach to behavior programming can be found in ConGolog [dLL00], where the generation and execution of plans are merged through a higher level of abstraction. ConGolog is based on Golog [LRL+97], a logic programming language based on Situation Calculus [MH69]. The main difference between Golog and ConGolog is the introduction in the latter of concurrency operators, that provide for a higher expressive power. The approach proposed by ConGolog aims at avoiding the computational burden associated to the general problem of planning, by restricting the search space. Instead of looking for a sequence of actions that would take the agent from an initial state to a goal state, in ConGolog a user defines a high-level program, which can be instantiated in sequences of actions to be executed. While action theories provide for several high level features that support plan design and verification, they require a substantial modeling effort and are not suitably supported by implementations. In this paper, we focus on Plan Design and, thus, we do not further discuss Plan Generation approaches.

In this paper, we describe a representation and execution framework for high level multi-robot behaviors, called *Petri Net Plans (PNP)* [ZIN+08, ZI06] that allows for effective and robust design of multi-robot systems. PNPs take inspiration from action languages (e.g., [Rei01]), and, thus, are explicitly defined as composition of actions.

The syntax and the semantics of PNPs is based on Petri Nets (PNs) [Mur89]. PNPs inherit from PNs many of their features, which are very useful in robotic application. One of the main advantages in using PNs is that they have an appealing and intuitive graphical representation. The (many) students involved in our projects, were able to design effective robotic behaviors with a minimal effort, using one of the many available graphical tools. Nevertheless, our experience in developing PNPs suggests that several types of errors occur frequently in plans. Consequently, we identify some properties of plans that, if verified, prevent from incurring into these errors. Furthermore, we show that verifying such properties
can be reduced to standard PN analysis problems and can be solved using standard PN analysis tools, thus enabling for debugging tools to support plan design.

PNPs are based on actions and operators. As any robotic system takes time to perform actions, we describe actions as non-instantaneous. This has a relevant impact on the language and allows for complex forms of execution control in terms of monitoring and failure recovery. Moreover, given that the environment is partially observable to robots, we model sensing actions, as a form of knowledge acquisition [SL93, DinR97]. Another relevant feature of many robots is that they can concurrently actuate several parts of their body. For example, a humanoid robot can use simultaneously, for different purposes, arms, legs and head. To this end, PNPs include operators to handle concurrent actions.

Multi-robot systems require robots to coordinate their actions in order to avoid interference and to perform complex tasks, that are not achievable by a single robot. To this end, we introduce coordination operators, which allow to ensure synchronization constraints among actions of different robots. In PNPs, one can specify a global model of the multi-robot system, where actions of different robots can be synchronized using direct communication. However, in order to avoid a central coordinator agent, which would introduce a single point of failure and a bottleneck for communication in the system, we provide a mechanism to automatically decompose the multi-robot PNP into a set of single-robot PNPs, which can be executed in a distributed fashion. We show that the properties that hold for the original centralized model, are still valid in the decomposed model of distributed execution, if the robots have access to a reliable communication channel.

Petri Net Plans can also be used for the implementation of cooperative behaviors. In particular, we show that in order to model cooperative behavior, coordination is not enough. Thus, we introduce a new operator, the multi-robot interrupt, which we use to model a general theory of teamwork, namely the Joint Commitment theory. As for coordination, we show that cooperative behaviors for PNPs allow for distributed execution.

The proposed framework has been implemented and is available\(^1\) as a C++ library and as an Open-RDK [CCIN08] module. PNPs have been tested on several robotic platforms (i.e., wheeled robots, quadruped robots AIBO and humanoid robots NAO) and in different domains (i.e., Search and Rescue [CFIN07], Soccer [ZIN+08] and Foraging [FINZ06]). A PNP implementation for the soccer domain obtained the Best Robotic Demo Award at AAMAS’08 [PZI+08]. In this paper, we present two case studies which show many of the relevant features of PNPs: complex coordination in a cooperative foraging problem and advanced cooperation using the Joint Commitment Theory in a soccer domain. Both case studies have been implemented on the AIBO robots.

Summarizing, the contributions of the proposed approach are manifold:

1. an intuitive approach to Plan Design inspired by action theories that, as a central feature, allows for automated plan verification;
2. a new model for monitoring the execution of robotic actions;
3. a formal model for cooperation and coordination in multi-robot systems;
4. an open source implementation of a development environment.

The paper is organized as follows. After presenting some related work in the next section, we define the basics of the PNP language in Section 3. Then, in Section 4 we provide an operational semantics in terms of an execution algorithm for PNP. In Section 5 we show some advanced features of the language for coordination which include operators for synchronizing actions of different robots and a distributed execution algorithm. Finally, in

\(^1\) available at: pnp.dis.uniroma1.it
Section 6, we show how PNPs can be used to achieve cooperation. We conclude by showing in Section 7 some of the multi-robot systems we have implemented and by discussing the features of the proposed approach in Section 8.

2 Related Work

This section provides an overview of the main approaches that have been proposed in the past few years for the representation and execution of robotic behaviors, providing the context in which Petri Net Plans have been developed. There is a wide literature about robot programming languages. Roughly, we can classify the approaches in two classes: FSA-based approaches and PN-based approaches.

The first class of approaches, i.e. FSA-based approaches, has been very popular in the recent years and stems from the need of implementing effective behaviors in real-time systems, where it is not possible to wait for the results of complex reasoning techniques [Gat92]. Several frameworks and implemented systems have been implemented based on this approach proving their effectiveness in real world applications (e.g., [Gat97]).

Although modeling behaviors based on FSAs is a very intuitive task, these approaches have been mostly limited to single-robot systems due to the lack of expressiveness in modeling concurrency. More recently, the scientific community has started to investigate the possibility to describe robotic behaviors using more expressive models. Probably, the most appealing approaches are the ones based on PNs. In particular, there has been a considerable effort in modeling MAS through PNs [Fer99] given their capability of representing concurrent systems and shared resources. PN based systems offer two main advantages with respect to FSA: 1) PNs provide a strictly more expressive framework compared with FSA and allow to represent any FSA in a more compact fashion; 2) PNs allow automatic analysis and thus can verify formal guarantees on the performance of the modeled systems. Nevertheless, there is currently no standard representation formalism for representing multi-robot systems based on PNs. Indeed, most approaches provide ad hoc solutions to specific problems.

2.1 FSA-based Approaches

Most of the robot programming languages developed up to now are based on Finite State Automata (FSA). FSA are either used explicitly, possibly supported by a graphical language, or they provide the underlying semantic model for the language.

PRS (Procedural Reasoning System [GL86]) is a general framework, inspired by the BDI (beliefs desire intentions) architecture. The beliefs in PRS are stored in a database, which an interpreter manipulates at run time to reach a set of goals (i.e. the desires). The knowledge on how to reach goals is stored as procedural knowledge, called Knowledge Areas. The knowledge areas are described by a graphical representation of an FSA, where edges are labeled with action names. Thus, actions are instantaneous events which determine state transitions in the FSA.

Colbert [Kon97] is a robot programming language which was developed as a component of the Saphira architecture [KMRS97]. Despite the fact that Colbert has a syntax which is a subset of ANSI C, its semantic is based on FSA. In particular, states correspond to actions while edges are events associated to conditions. Moreover, Colbert allows some simple form of concurrency although, in this case, the semantics is considerably different from standard FSA semantics and it is very hard to guaranty coherence in the behaviors. Probably, the
most interesting feature associated to concurrency in Colbert is the possibility to monitor and interrupt actions.

The limitations of FSA have raised the issue of finding more expressive formalism to control robots. Some approaches, such as ESL [Gat92], address the problem by defining constructs commonly used in robotics, without limiting the expressiveness of the programming language which is based on Lisp. In a similar way, the Task Description Language (TDL) [SA98] extends C++ in order to include asynchronous constrained procedures, called Tasks. TDL programs have a hierarchical structure, called Task Tree, where each child of a given task is an asynchronous process and execution constraints among siblings are explicitly represented. Xabsl [LRJ06] is a more recent approach, mainly developed in the frame of the RoboCup competition and is somehow similar to TDL. This approach is based on a hierarchical structure of behaviors and is bundled with a set of highly engineered tools which allow efficient development of behaviors. In ESL, TDL and Xabsl no analysis of the resulting behavior is possible and coding coherent behaviors requires a considerable modeling effort.

The Reactive Action Packages (RAPs) [Fir89] are the robot programming language of the Animate Agent Architecture [FPS98]. RAPs are expressed in lisp-like syntax and describe concurrent tasks along with execution constrains. RAPs are an ad-hoc tool to execution of concurrent tasks in robotic applications which have some similarities with PNs. Nevertheless, it is not possible to perform analysis of RAPs, mainly because there is no underlying formal model.

Although FSA-based approaches, have been very successful in modeling many single-robot systems their expressive power limits their applicability to multi-robot systems. In general, more expressive formalisms are required in order to model the inherent concurrency of multi-robot systems. The extensions of FSA-based approaches to handle concurrency usually have ad-hoc semantics and do not allow formal analysis, making it difficult to develop robust and effective behaviors.

2.2 PN-based Approaches

In the past few years, approaches to plan generation and representation based on Petri Nets, have gained increasing interest. PNs provide a mathematical and graphical framework for the representation of discrete event systems suitable to model the behavior of a wide range of Discrete Event Systems (DES), including robotic systems. One desirable advantage provided by PNs to this extent is the possibility to provide formal properties of the produced models, using some standard analysis methods. We here report some of the works that have been recently published on the representation of robotic behaviors using PNs.

Petri nets have been used to model multi-agent systems. In [CDG07], the authors present a framework to build multi-agent systems using Petri nets, that allow to perform static analysis to assess some important properties of the system such as deadlock avoidance. The model is limited to purely reactive agents: actions are instantaneous, represented by Petri net transitions, and the places of the Petri net model represent the environmental state of the agent. The assumption of instantaneous actions does not allow to adequately model robotic systems where actions have a duration and must be monitored during execution. Similar considerations apply to [BDK01], where the authors present a Petri Net Algebra, which allows the design of Multi-Agent Systems using a component-model approach, based on composition operators. In this approach, individual and system goals are specified as coverable markings in the Petri Net representing the MAS. Interestingly, the proposed methodology is
property-preserving, and ensures that formal characteristics of the net are maintained in the composition of the multi agent system.

There has been other work which addresses specifically robotic systems. Nevertheless, these approaches develop ad-hoc models for specific applications, rather than providing a general language. For example, in [SY05] PNs are used to model a multi-robot coordination algorithm, based on an auction mechanism, to perform environment exploration. Similarly, [XVIY02] shows an agent-based extension of Fuzzy Timed Object-Oriented Petri Nets (proposed in [MM01]) for the design of cooperative multi-robot systems for a specific industrial application. Another example is [KL06], where the authors report the use of distributed agent-oriented Petri Nets for the modeling of a Multi-Robot System for soccer playing.

Other PN-based approaches, rather than modeling a behavior, model the robotic system and then try to optimize it in order to produce the desired behavior. In [Mil02], the authors propose an approach for modeling single-agent systems. In this case, users define several possible implementations of a task. Then, a reinforcement learning algorithm is used to select an optimal solution. This approach also exploits formal analysis of the PN models allowing for qualitative evaluation (i.e. stability, controllability and possibility of error recovery). This single-agent approach has been tested in two real world applications. A similar work is presented in [CL07], but in this case the PN model includes explicitly a representation of the environment. A more general approach, able to model multi-robot systems, can be found in [KPG03]. In this work, plans for each single robot are generated either using a graphical interface or using some automated planning method. The plans are then compiled into Petri nets for analysis, execution, and monitoring of their joint execution. The operators that are used for the PN representation of the plans are inspired by the STRIPS [FN71] planning system. Supervisory control techniques are applied to the Petri net controller in order to identify possible conflicts that may arise due to the presence of shared resources among the multiple robots. To deal with unforeseen events replanning is used at run-time severely limiting the applicability of this approach to real-time systems in dynamic environments.

The approach that we propose in this paper, Petri Net Plans, is a formalism to represent behaviors that allows for formal verification of plans. Opposed to other formalisms, PNP s provide a general framework for describing multi-robot behaviors. PNP s are more expressive than most approaches in literature, allowing for complex forms of coordination and cooperation. PNP s are very intuitive and require a small modeling effort, thanks to the explicit characterization of atomic structures which are explicitly interpreted as actions and operators to combine actions. The approach has been tested in many application domains and has been shown to be particularly appropriate for real-time systems acting in dynamic environments.

3 Petri Net Plans Language

PNPs allow for specifying plans describing complex behaviors for mobile robots. These plans are defined by combining different kinds of actions (ordinary actions and sensing actions) using control operators. Thus, in this section, we provide a description of PNP s through actions and operators, initially ignoring multi-robot operators, which are used to achieve coordination and cooperation. Thus, we can consider the multi-robot system, as composed by robots which execute their own PNP independently from others. Nevertheless, as we will see in Sections 5 and 6, PNP s naturally model multi-robot systems when enriched with multi-robot operators.
PNP Language (PNPL) is a high-level robot programming language that builds upon ordinary PNs and provides the syntax for PNPs. In PNPL, places are allowed to have at most one token and edges all have a weight of one. The PNP language is based on actions, which are models of primitive behavior execution. In this section, we provide a detailed description of the interpretation of PNP. More details on the (operational) semantics of PNPs are given in Section 4.

A PNP is a PN \( \langle P, T, F, W, M_0 \rangle \) with a domain specific interpretation and an extended semantics.

Formally, in the tuple \( \langle P, T, F, W, M_0 \rangle \):
- \( P = \{p_1, p_2, \ldots, p_m\} \) 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.
- \( W : F \rightarrow \{1\} \) is a weight function for edges. In PNPs \( w(f_s, f_d) = 1 \) for each pair \( f_s, f_d \in F \).
- \( M_0 : P \rightarrow \{0, 1\} \) is the initial marking, denoting the initial state of the robot.
- \( P \cup T \neq \emptyset \) and \( P \cap T = \emptyset \)

In a PNP, nodes have different interpretations, thus they are partitioned into four classes:
- \( P = P^I \cup P^O \cup P^E \cup P^C \), where:
  1. \( P^I \) is the set of input places, which model initial configurations of the PNP;
  2. \( P^O \) is the set of output places, which model final configurations of the PNP;
  3. \( P^E \) is the set of execution places, which model the execution state of actions in the PNP;
  4. \( P^C \) is the set of connector places, which are used to connect different PNPs.

Also transitions are partitioned in three subsets \( T = T^S \cup T^T \cup T^C \), where:
- \( T^S \) is the set of start transitions, which model the beginning of an action/behavior;
- \( T^T \) is the set of termination transitions, which model the termination of an action/behavior;
- \( T^C \) is the set of control transitions, which are part of the definition of an operator.

A PNP models the execution of actions by using specific places that represent the execution states of the related behaviors. In particular, each action has an execution place \( e \in P^E \) and \( M(e) \) determines whether the behavior is active or not. Thus the set of execution states of a PNP (i.e., \( P^E \)) models the execution state of the system and its temporal evolution characterizes the system dynamics. Moreover, we define an initial marking \( M_0 \), which specifies the initial execution state of a PNP, and a goal marking \( G \), which specifies the set of desired termination states. The initial and goal markings can be in general any marking defined on the non-executable places of a PNP.

In the remainder of this section, we describe PNPs only in terms of their structure, thus ignoring their markings. Moreover, we will omit \( W \), since it is constantly set to 1. Therefore, we consider a generic PNP as the tuple \( \langle P, T, F \rangle \) and, for the sake of readability, we present the topological structure of nets (i.e., \( F \)) through a graphical notation.

In order to define complex PNPs, we first describe the most elementary PNP actions and then show how to combine them through operators.

### 3.1 Actions

Actions represent primitive behaviors of robots and are the atomic concept upon which we build complex PNPs. Actions are characterized by having \( T^C = \emptyset \). There are two types of actions: 1) ordinary action and 2) sensing action.
Ordinary-action. This elementary structure models a deterministic action. It explicitly represents the action as non-instantaneous, by defining its start event $t_s$, execution state $p_e$, and termination event $t_e$. An ordinary action is a PNP with the structure shown in Figure 1 (a) where:

- $P_I = \{p_i\}$, $P_E = \{p_e\}$ and $P_O = \{p_o\}$,
- $T_S = \{t_s\}$ and $T_T = \{t_e\}$.

Sensing-action. Sensing actions are a special kind of non-deterministic actions, where the actual outcome of the action depends on some property which may be known only at execution time. A sensing action is a PNP with the structure shown in Figure 1 (b) in which:

- $P_I = \{p_i\}$, $P_E = \{p_e\}$, $P_O = \{p_{o_1}, p_{o_2}\}$,
- $T_S = \{t_s\}$ and $T_T = \{t_{e_1}, t_{e_2}\}$.

where $t_{e_1}$ and $t_{e_2}$ are the transitions ending the action when the sensed property is true, and when it is false, respectively. Analogously, the places $p_{o_1}$ and $p_{o_2}$ terminate the action when the sensed property is true and when it is false. Notice that, it is easy to extend sensing actions in order to have more than two mutually exclusive outcomes. We sometimes consider an instantaneous variant of sensing actions (i.e., without $t_s$ and $p_e$), which we call evaluation action. Evaluation actions are used to query the knowledge of the robot, and do not require to act in the real world.

3.2 Operators

PNPs are modular, since they allow for combining multiple nets in order to build complex structures. Two nets can be combined by merging two places, one for each net. This allows for sequencing behaviors and constructing loop structures. Moreover, it is possible to monitor the execution state of a PNP by using interrupt operators, that tie the execution places of a net with the input places of another net through a transition (interrupt) that when verified suspends the execution of the current action and triggers a recovery plan. This feature is very useful when dealing with robots in dynamic situations, where failure recovery is a fundamental issue. Finally, concurrency operators are used to model concurrent behaviors in a single robot, or multi-robot actions.

Therefore, in order to create complex PNPs, three kinds of operators are defined: sequence, interrupt, and concurrency operators. Operators are PN used as control structures.
which do not refer to specific behaviors. Operators are thus characterized by having $P^E = \emptyset$, $T^S = \emptyset$, and $T^T = \emptyset$.

**Sequence Operator.** Sequence operator is obtained by merging two places of two different PNPs into one. For example, an output place of a first PNP can be merged with an input place of a second PNP to obtain a chain of the two PNPs. The sequence of a motion behavior for approaching a ball (i.e., gotoball) and a kicking behavior are shown in Figure 3. The dashed circle shows the result of the merge of the output place of the gotoball action and the input place of the kick action.

Formally, given two PNP structures $PN_1 = \langle P_1, T_1, F_1 \rangle$ and $PN_2 = \langle P_2, T_2, F_2 \rangle$ ($PN_1$ can be either different or equals to $PN_2$), a non-execution place $p \in P^I_1 \cup P^D_1 \cup P^C_1$ and an output place $o \in P^O_2$, the sequence of $PN_1$ and $PN_2$ obtained by merging $p$ with $o$ is a PNP structure $PN = \langle P, T, F \rangle$, with $P = P_1 \cup P_2 - \{o\}$, $T = T_1 \cup T_2$, $F = F_1 \cup F_2 - \{(t_e, o)\} \cup \bigcup_{t_e \in \bullet o} \{\langle t_e, p \rangle\}$, where $\bullet o$ is the set of input transitions of $o$. Moreover, $P^I = P^I_1 \cup P^I_2 - \{p\}$, $P^D = P^D_1 \cup P^D_2 - \{o\}$, $P^C = P^C_1 \cup P^C_2$, $P^E = P^E_1 \cup P^E_2$ otherwise.

This operator can be applied to two places of the same PNP for creating loops. Thus, when $PN_2 = PN_1$, we call this operator loop-sequence.

**Interrupt Operator.** The interrupt operator, shown in Figure 2 (a), is a very powerful tool for handling action failures. In fact, it can interrupt actions upon failure events and activate recovery procedures. In the example shown in Figure 4, the gotoball action is monitored by an interrupt triggered by when the ball is stolen from an opponent. As a recovery procedure,
The interrupt of an action and its recovery procedure.

Formally, given two PNP structures $PN_1 = \langle P_1, T_1, F_1 \rangle$ and $PN_2 = \langle P_2, T_2, F_2 \rangle$ ($PN_1$ can be either different or equals to $PN_2$), an execution place $e \in P_1^E$ and a non-execution place $p \in P_2^I \cup P_2^O \cup P_2^C$ ($P_2$ can be either different or equal to $P_1$), the interrupt of $PN_1$ through $PN_2$ obtained by merging $e$ and $p$ is a PNP structure $PN = \langle P, T, F \rangle$, with $t_{int}$ is the transition associated to the interrupt condition. Moreover, $P = P_1 \cup P_2$, $T = T_1 \cup T_2 \cup \{t_{int}\}$, $F = F_1 \cup F_2 \cup \{(e, t_{int})\} \cup \{(t_{int}, p)\}$, where $t_{int}$ is the transition associated to the interrupt condition. Moreover, $P^O = P_1^O \cup P_2^O$, $P^E = P_1^E \cup P_2^E$, $P^C = P_1^C \cup P_2^C \cup \{p\}$, if $p \in P_1^I$, while $P^C = P_1^C \cup P_2^C$ otherwise. Finally, $T^C = T_1^C \cup \{t_{int}\}$.

Interrupt is often used to go back to a previous part of the plan in order to re-try the execution of a portion of it. In these cases, when $P_2 = P_1$, we call this operator loop-interrupt.

Many robotic systems require to handle concurrency due to: 1) the possibility of actuating simultaneously, and independently several parts of the body and 2) the possibility of distributing the system into multiple robots. In the following, we deal with the first issue, while we provide a discussion of multi-robot distributed execution, along with the definition of appropriate operators, in Sections 5 and 6. In particular, here we present fork and join operators for dealing with multiple actuators.

**Fork Operator.** Each token in a PNP can be thought as a thread of execution. The fork operator generates from a single thread of execution, multiple threads. Figure 2 (b) shows a fork structure for producing two threads of execution. The operator can be easily extended to generate more threads by adding new output places. The fork operator is characterized by $T^C = \{t_f\}$, $P^I = \{p_i\}$ and $P^O = \{p_o_1, p_o_2\}$.

**Join Operator.** The join operator allows to synchronize multiple threads of execution. This operator consumes multiple threads of execution simultaneously, and generates a single synchronized thread. The join structure is shown in Figure 2 (c), for the case of two threads. As in the previous case, the operator can be easily generalized to synchronize more threads by adding new input places. The join operator is characterized by $T^C = \{t_j\}$, $P^I = \{p_i_1, p_i_2\}$ and $P^O = \{p_o\}$.

Formally, the sequence of PNPs with the PN structures denoting fork and join returns a new PNP structure. Figure 5 shows the sequence of a fork operator and two actions (i.e., gotoBall and trackBall). The fork transition generates two threads of execution (i.e., tokens) which concurrently execute until the robot reaches the ball. Finally, the two threads are joined by the join operator. The dashed circles show the result of merging: the output places
of the fork operator with the input places of the actions; and the output places of the actions with the input places of the join operator.

3.3 Syntax of PNPs

The syntax of a PNP can be defined through the inductive closure of basic actions, under the sequence, interrupt, fork and join operators. More formally, a precise characterization of PNP syntactic structure is given by the following definition.

Definition 1 The PNP structure described by PNPL is defined as:

- an ordinary or a sensing action
- the sequence of two PNPs
- the loop-sequence of a PNP
- the interrupt of a PNP with another PNP
- the loop-interrupt of a PNP
- the sequence of fork operator with two PNPs
- the sequence of join operator with two PNPs

Notice that these structures do not yet have all the features needed for execution. Therefore in the following section we will define additional properties that must be respected in order to fully characterize PNPs.

3.4 PNP Definition

In order to define PNPs that are actually executable, PNP structures defined above must fulfill some additional requirements that are defined in the following.

Tokens of PNPs are defined as execution threads, which activate the execution of atomic behaviors, represented by actions. A first important property is to enforce that the number of execution threads is bounded, in the sense that for any possible execution state there is no more than a token in each place. If this is not the case, it could be that multiple execution threads control the same atomic action. This is an undesirable situation because the semantics of PNPs assumes that each action is an atomic behavior, thus controlled by a single thread of execution. Nevertheless, there is no guarantee that any PNP respects this constraint during execution.
Definition 2 A PNP structure \( \langle P, T, F, W, M_0, G \rangle \) is safe if any reachable marking \( M \) satisfies:
\[
\forall p_i \in P \quad M(p) \leq 1
\]
that is, the net is 1-bounded.

In PNs, this property is called 1-boundedness, and can be automatically verified through standard analysis techniques (e.g., coverability tree).

Another common requirement is that any transition defined in the PNP is not dead, in the sense that there exists a sequence of markings from \( M_0 \) such that the transition is fired at least once. Clearly if there exists a transition which can never be fired, there is something wrong in the plan, e.g., because there is a dead-lock in the net. Thus, we want a minimal PNP, in the sense that each transition in the net is necessary and, thus, can be fired at least once in some sequence of markings. This property corresponds to \( L_1 \)-liveness in PNs.

Definition 3 A PNP structure \( \langle P, T, F, W, M_0, G \rangle \) is minimal if it is \( L_1 \)-live.

In PNs, this property (\( L \)-liveness), and can be automatically verified through standard analysis techniques (i.e., liveness analysis).

Finally, notice that, as a structural difference with PNs, PNPs have a set of goal markings which describe the success of the plan. In this case, it makes sense to execute a plan if the goal is reachable from any reachable state. In PNs, such goal state is called a home state. If the goal state is not a home state, the plan execution could prescribe useless actions or get stuck in a dead-lock.

Definition 4 A PNP structure \( \langle P, T, F, W, M_0, G \rangle \) is effective, if the goal marking is a home state.

In PNs, the property that the goal state is a homing state can be automatically verified through standard analysis techniques (i.e., reachability analysis).

Based on the previous considerations, we constrain PNPs to be safe, minimal and effective.

Definition 5 A PNP is a PNP structure that is
- safe (Definition 2)
- minimal (Definition 3)
- effective (Definition 4)

PNPs as defined above are actually executable by the execution algorithm described in the next section.

Sub-Plans

In the design of a PNP, sub-plans can be used for a higher modularity and readability. A sub-plan is represented as an ordinary action but refers to a PNP rather than to a primitive behavior. A plan execution module, running on the robot, takes care of dynamically loading sub-plans in case a super-plan invokes its execution. In particular, whenever a start transition of a sub-plan is fired, the marking of the sub-plan is set to the initial one. The sub-plan will then be executed, possibly concurrently with other primitive behaviors or sub-plans, until it reaches its goal marking or a condition labeling its ending transition is met. Moreover,
sub-plans allow for a more powerful use of interrupts, which can be used to inhibit a whole complex behavior (i.e., a sub-plan) at once.

A complete discussion of PNP s with sub-plans is out of the scope of this paper. Nevertheless, the discussion which follows can be generalized in many cases to PNP s with sub-plans. In the simplest case where sub-plans are used as macros, a PNP with sub-plans can be transformed to a PNP without sub-plans by recursively unfolding sub-plans. In the following, we discuss only PNP s without sub-plans.

3.5 Robotic Soccer Example

Consider a soccer robot, which must find a ball in a soccer field, reach it and then shoot. In this simple example we assume that, initially, the ball is not far away from the robot. The PNP in Figure 1(a) shows a possible PNP for this behavior. The robot starts to seek for the ball using a sensing action. Notice that a branch of the sensing action, is closed in a loop. Thus, the robot will continue to seek for the ball until it finds it. Then, the PNP has a fork operator, in order to: 1) approach the ball by actuating the legs of the robot and 2) track the ball with the head. Both actions are monitored by interrupt operators. In one case, if the tracking behavior looses visual contact with the ball, it will rollback to the seek behavior. In the other case, if someone moves the ball far away from the robot, it will rush to get again close to it. Finally, once the robot has reached the ball, the two actions join, and then the robot shoots the ball towards the opponent.

The analysis of such plan brings about some issues. First, the net is not safe. If the robot sees the ball, but this is far away, it will rush and correctly continue to track the ball. Nevertheless, once he is closer to the ball, the approach ball and the track ball behavior will both receive an execution token, thus resulting in two tokens in the trackball behavior. Moreover, the PNP is also not minimal. In particular, the join transition can never be fired because it has an input place which is a source, and has no tokens in the initial marking. In this specific case, this also means that the PNP is not effective, because the goal marking can never be reached. These issues can all be detected with PN analysis tools. A valid PNP for this behavior is shown in Figure 1(b).

4 Execution

PNPs are a subclass of PNs aimed at describing the execution of a robotic behavior. As such, we consider the marking of the PN as the execution state of the behavior. The operational semantics of PNP s, as for any PN, is defined by the firing rule. The firing rule describes the dynamics of the system based on events, which are generally model dependent. That is, events, specific to the model at hand, determine the actual firing of enabled transitions.

The main difference with respect to PN, is in the way PNP s interpret events and transitions: we can distinguish among controllable and non-controllable transitions, depending on whether the controller (i.e., executor) of the plan can control or not the related events. Non-controllable transitions usually depend on external events, while controllable ones depend on control strategies.

The only controllable transitions in PNP s are the ones which correspond to action starts. In this paper, we assume that plans do not have non-deterministic choices other than sensing actions. In this case, we can adopt a very simple control strategy, which states that all the controllable transitions, when enabled, must fire. Nevertheless, the proposed approach
Table 1: A robotic soccer example: striker robot. The labeling shown is an example of the actual syntax used for our executable plans. Labels characterize action nodes and conditions associated to transitions. The net (a) denotes a PNP structure which is not a valid PNP, while net (b) is a valid PNP.
is applicable also if an appropriate control strategy is adopted when plans contain non-deterministic choices (e.g., GOLOG like).

In PNPs, non-controllable transitions are those which depend on observable properties of the environment. For example, a robot could fire a transition (e.g., interrupting a gotoball behavior), if it looses visual contact with the ball. In order to specify external events for non-controllable transitions, we define a labeling mechanism. In particular, all non-controllable transitions may be labeled with conditions to be verified in order for the related event to occur. A condition $\phi$ on the transition $t$ is denoted with $t.\phi$. If no condition is specified for a non-controllable transition, we will consider it to be True. We assume that the actual knowledge of a robot is accessible for the executor through a knowledge base $kb$. During the execution of a plan, we determine whether a given external event occurs by querying its local knowledge base $kb$. That is, an enabled transition $t$ fires if $kb \models t.\phi$.

During the execution of a PNP the knowledge base of the robot could change, due to the acquisition of new knowledge. In particular, we characterize the evolution of a PNP through a sequence of markings and knowledge bases $\langle M_i, kb_i \rangle$, representing the execution state and the knowledge of a robot at time $\tau_i$.

**Definition 6 (Evolution)** An Evolution of a PNP $P = \langle P, T, F, W, M_0, G \rangle$ is a temporally annotated sequence of pairs $\langle (M_0, kb_0), \ldots, (M_n, kb_n) \rangle$. In particular, each $\langle M_i, kb_i \rangle$ represents a marking $M_i$ obtained, given a knowledge base $kb_i$, at time $\tau_i$.

An Admissible Evolution of a PNP $P$ is a sequence of markings which can be obtained evolving $P$, from the initial marking, according to the semantics of events in PNP.

**Definition 7 (Admissible Evolution)** An Admissible Evolution of a PNP $P = \langle P, T, F, W, M_0, G \rangle$ is an evolution

$\langle (M_0, kb_0), \ldots, (M_n, kb_n) \rangle$

where $M_0$ is the initial marking, $\forall i < n$ $M_i \notin G$ and such that:

$\forall i \in \{0..n-1\} \exists t \in T : enabled(t, M_i) \land fire(M_i, t) = M_{i+1} \land kb_{i+1} \models t.\phi$

### 4.1 Abstract Robot Architecture

For the sake of clarity, we describe the execution algorithm for PNPs based on an abstract robot architecture shown in Figure 6. Nevertheless, the use of PNPs is not restricted to this particular type of architecture (e.g., could be used in subsumption architectures too [Bro91]). In particular, we define a two layer architecture:

- **Symbolic Layer**: composed by a PNP library, a knowledge base and a PNP executor.
- **Numeric Layer**: composed by data fusion modules and low level robotic behaviors.

**Symbolic Layer.** The symbolic layer consists of the PNP executor, which implements the PNP execution algorithm, and the Knowledge Base, which maintains the current information on the environment. The evolution of the plan must be controlled according to the robot’s actual knowledge (i.e., according to its epistemic state of knowledge), since we can not assume that the robot has complete knowledge about all the properties of the environment. The knowledge base can be implemented in any any formalism: for example, in our implementation we use a simple conjunction of propositions. Analogously, queries $\phi$ 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 robot is able to evaluate queries over the current model of the world, i.e., to compute $kb \models t.\phi$. 

Fig. 6 An Abstract Robot Architecture for PNPs

**Numeric Layer.** In order to effectively interpret noisy and unreliable sensor data, we assume that our robot can use standard numeric approaches for data fusion [TBF05], such as localization, mapping, tracking, etc. Notice that this numerical information must be anchored to the symbols in the knowledge base [CS03]. In order to effectively control the behavior of the robot we assume the availability of a set of implemented actions \( A = \{ a_1, \ldots, a_k \} \). According to the specification on PNPs, each action considered here is an abstraction for the implementation of a specific behavior that the robots can execute. We assume that actual behavior execution will be performed in a separate thread with respect to the PNP Executor\(^2\). This means that after an action is started, it will remain active until either end or interrupt will be invoked.

4.2 PNP Execution Algorithm

Algorithm 1 is the execution algorithm for PNPs. As previously described, it relies on a KB for evaluating events related to non-controllable transitions and on a set of implemented actions, which can be controlled through the start(), end() and interrupt() procedures. The main 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 \( \{ M_0, \ldots, M_n \} \) that,

\(^2\) Obviously, this can be easily extended to non-threaded cases.
Algorithm 1 PNP Execution Algorithm

Domains:
\[ A = \{a_1, \ldots, a_k\} \]: Set of Implemented actions
\[ \Phi \]: Set of terms and formulas about the environment
\[ TrType = \{\text{start}, \text{end}, \text{interrupt}, \text{standard}\} \]

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

Global Variables:
KnowledgeBase : \( kb \)

procedure execute(PNP \( \langle P, T, F, W, M_0, G \rangle \))
1: CurrentMarking = \( M_0 \)
2: while \( \text{CurrentMarking} \not\in G \) do
3: for all \( t \in T \) do
4: if enabled(\( t, \text{CurrentMarking} \)) \&\& \( kb|_t.\phi \) then
5: handleTransition(\( t \))
6: \( \text{CurrentMarking} = \text{fire}(\text{CurrentMarking}, t) \)
7: end if
8: end for
9: end while

procedure handleTransition(\( t \))
if \( t.t = \text{start} \) then
   \( t.a.\text{start}() \)
else if \( t.t = \text{end} \) then
   \( t.a.\text{end}() \)
else if \( t.t = \text{interrupt} \) then
   \( t.a.\text{interrupt}() \)
end if

possibly, evolve the system from the initial marking \( M_0 \) to a goal marking \( M_n \in G \). In particular, at each step, Algorithm 1 checks (line 4) if each transition \( t \in T \) is enabled \( (\text{enabled}(\text{CurrentMarking}, t)) \) and if the related event occurs. If these two conditions are satisfied, the transition \( t \) is fired (line 6) and the corresponding 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 1 (Correctness) Algorithm 1 correctly executes any PNP \( P \), i.e.:
- any computed evolution \( (\langle M_0, kb_0 \rangle, \ldots, \langle M_n, kb_n \rangle) \) of \( P \) is an admissible evolution
- the behaviors of a robot are started, interrupted or ended, when the the start, the end and interrupt transitions of the corresponding actions are fired;

Proof In order to prove that Algorithm 1 correctly executes a PNP we must show that:
1. any evolution is admissible
2. behaviors are controlled according to the semantics of PNP.

The second part of the proof is trivially obtained by the procedure handleTransition(\( t \)), we thus focus on the first. To prove that the evolution is admissible we need to show that it
obeys to the firing rule (i.e. enabled(t, M_i) \land fire(M_i, t) = M_i+1), and that it evaluates correctly conditions labeling transitions (kb_i+1 \models t.\phi). The former requirement is explicitly satisfied by lines 4 and line 6, while the latter by line 4.

\qed

5 Coordination using PNPs

In the following, we show how PNPs can be used to support coordination so to avoid interference among robots and enhance the performance of the system through the use of joint actions. In the literature, 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 robots executing them, or as plan merging, where individual plans are merged into a multi-robot plan (see [Dur99] for details). In our work, we follow the centralized planning approach that has been easily implemented in our formalism as described in this section. Moreover, we provide a distributed execution model by implementing a centralized planning for distributed plans approach [Dur99]. Our 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 robot. Assuming that robots have access to a reliable communication channel, the correctness of the distributed execution, with respect to the multi-robot PNP, is enforced using communication primitives.

5.1 Synchronization Operators

We can consider a multi-robot PNP, for robots R_1, ..., R_n, as the union of n PNPs (one for each robot). In a multi-robot PNP, each element of the net is labeled with the unique name of the robot which is in charge of its execution. This will be indicated in the following with the prefix R_j: for example, R_j.P refers to the set of places associated to robot R_j. Thus, given n single-robot PNPs \{⟨ R_j.P, R_j.T, R_j.F ⟩ \}_{j=1}^n we define a multi-robot PNP as:

\[ \mathcal{P} = ⟨ P, T, F ⟩ \]

where \( P = \bigcup_{j=1}^n R_j.P \), \( T = \bigcup_{j=1}^n R_j.T \), \( F = \bigcup_{j=1}^n R_j.F \).

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 robots (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. Therefore, we want to be able to order actions across plans so that overall consistency is maintained and conflicting situations are avoided.

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 (h_sync), that synchronizes two plans at a given point in time, and a soft synchronization operator (s_sync), which introduces a precedence relation among the actions of two plans.

**Hard Synchronization Operator.** The hard synchronization operator (h_sync), shown in Figure 7, supports time synchronization for the actions of two robots R_1 and R_2. The operator has two input places \( P^I = \{ p_{i1}, p_{i2} \} \) and two output places \( P^O = \{ p_{o1}, p_{o2} \} \) and
formally, consider two robots \( R_1 \) and \( R_2 \), four PNPs \( R_1.P_1, R_1.P_2, R_2.P_1, \) and \( R_2.P_2 \), and the \( \text{h}_\text{sync} \) operator. The PN formed by the sequence of \( R_1.P_1 \) and \( \text{h}_\text{sync} \) through the place \( p_{i1} \), the sequence of \( R_2.P_1 \) and \( \text{h}_\text{sync} \) through the place \( p_{i2} \), the sequence of \( R_1.P_2 \) and \( \text{h}_\text{sync} \) through the place \( p_{o1} \), and the sequence of \( R_2.P_2 \) and \( \text{h}_\text{sync} \) through the place \( p_{o2} \) is a multi-robot PNP.

\( T^C = \{t\} \). The operator is similar to a join and a fork, except that it is used to synchronize behaviors of different robots.

Example 1 Figure 9(a) shows a PNP for two robots which have to lift a table by grabbing it at two opposite sides. The nodes for action structures and synchronization operators are grouped, for readability, by a common label. In this example, \( R_1 \) and \( R_2 \) can reach the two sides of the table asynchronously, but have to lift it simultaneously. The \( \text{h}_\text{sync} \) operator ensures that the robots will start to lift the table when both have reached it. In particular, the input transition \( t \) acts as a join waiting for both actions \( R_1.\text{gotoLeftSideTable} \) and \( R_2.\text{gotoRightSideTable} \) to terminate. When both actions have terminated, the transition \( t \) acts like a fork enabling the simultaneous execution of the lift actions.

Soft Synchronization Operator. The soft synchronization operator (s_sync), shown in Figure 8, can be used to force a precedence relation among the actions of two different robots. The operator has two input places \( P^I = \{p_{i1}, p_{i2}\} \) and two output places \( P^O = \{p_{o1}, p_{o2}\} \) and \( T^C = \{t_f, t_j\} \).

Formally, consider two robots \( R_1 \) and \( R_2 \), four PNPs \( R_1.P_1, R_1.P_2, R_2.P_1, \) and \( R_2.P_2 \), and the \( \text{s}_\text{sync} \) operator. The PN formed by the sequence of \( R_1.P_1 \) and \( \text{s}_\text{sync} \) through the place \( p_{i1} \), the sequence of \( R_2.P_1 \) and \( \text{s}_\text{sync} \) through the place \( p_{i2} \), the sequence of \( R_1.P_2 \) and \( \text{s}_\text{sync} \) through the place \( p_{o1} \), and the sequence of \( R_2.P_2 \) and \( \text{s}_\text{sync} \) through the place \( p_{o2} \) is a multi-robot PNP.

Example 2 Figure 10(a) shows an example of the use of the \( \text{s}_\text{sync} \). In this example, there are two robots \( R_1 \) and \( R_2 \). The first robot, has is a mail delivery robot and has a manipulator, while the second is a vacuum cleaner robot and has no manipulation ability. The first robot opens the door of the room to be cleaned and then moves on to deliver the mail. The second
robot moves to the door and the enters the room to clean it. The problem is that the second robot has to be sure that the door is open before entering the room. To this end, we can add a \textit{ssync}. This allows the first robot to notify the second that the door is open, without having to wait for the second robot to reach the door. On the other hand, the second robot, when received the notification, can go on and enter the room safely.

5.2 Distributed Execution

The semantics of a multi-robot PNP is the same of a single-robot PNP in the case of multi-body planning [RN03], where a single centralized agent can dictate actions prescribed by the plan and query the knowledge base of each robot. 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. Specifically, 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.

In particular, we aim at reproducing behavior of synchronization operators in a distributed way, by replacing the synchronization operators with appropriate sequences of non-blocking send actions and blocking receive actions. We represent the two ordinary actions as \textit{receive}(R_s, id) and \textit{send}(R_r, id), where \(R_r\) is the robot that receives the message sending the message \(id\), while \(R_s\) is the robot sending it. \(id\) is unique identifier for the synchronization operators (which can be obtained by enumerating the operators). Notice that, this procedure can be viewed as a distributed version of the firing rule, relatively to synchronization structures.

Let us start with an example of use of \textit{hsync} shown in Figure 9(a). The multi-robot plan can be decomposed easily into two single robot plans as shown in Figure 9(b), where
the top plan belongs to $R_1$ and the bottom one to $R_2$. The plans look much the same, except that the $h_{\text{sync}}$ has been replaced by two ordinary actions.

The joint execution of the two plans leads to the same behavior of the multi-robot plan, when executed centrally. To this end, the behavior of the communication primitives $\text{send}(r, id)$ and $\text{receive}(r, id)$ has a key-role. Assume, without loss of generality, that $R_2$ reaches the table first. This means that it will perform a non-blocking $\text{send}(R_1, id)$, and then stop on the blocking receive $\text{receive}(R_2, id)$. When $R_2$ arrives on the other side of the table, it will perform a non-blocking $\text{send}(R_2, id)$. At this point, both robots will have received the $id$ message, and will move on lifting the table together.

Consider now the example of $s_{\text{sync}}$ shown in Figure 10(a), and the two plans derived from it in Figure 10(b). In this case, the $s_{\text{sync}}$ operator $s_{\text{sync}}(R_1, R_2)$, is decomposed in only two primitives: a non-blocking send and a blocking receive. The reason for this asymmetric behavior is that the $s_{\text{sync}}$ has a different behavior depending on the robot. In particular, the robot $R_1$ needs only to send a message $id$ to robot $R_2$, that states that it accomplished his task (i.e., opening the door). The send message is non-blocking because there is no need to wait for executing $\text{enterRoom}$. Nevertheless, robot $R_2$ needs to wait for the message, and thus, must perform a blocking receive $\text{receive}(R_1, id)$ on the main thread, because it has to be sure that $\text{openDoor}$ has ended, before performing $\text{enterRoom}$.

**Distributed Execution Algorithm**

We can, thus, decompose a PNP $\mathcal{P}$ into several nets $\langle R_1.\mathcal{P}, \ldots, R_n.\mathcal{P} \rangle$, one for each robot. Each $R_i.\mathcal{P}$ can be executed locally on robot $R_i$ and asynchronously with respect to other robots. The exchange of messages among the robots allows to coordinate the behavior of each $R_i.\mathcal{P}$.

**Definition 8 (Distributed Execution Algorithm -DEA-)** Given a PNP $\mathcal{P}$ for $n$ robots, and its decomposition $\langle R_1.\mathcal{P}, \ldots, R_n.\mathcal{P} \rangle$, the distributed execution of $\mathcal{P}$ consists of the parallel execution of each $R_i.\mathcal{P}$ according to Algorithm 1.
Fig. 10 (a) A multi-robot PNP using \( s_{\text{sync}} \). (b) The single-robot PNPs obtained from the multi-robot one.

Notice that the execution of each \( R_i.P \) is accomplished as described in Section 4; the main difference is that the execution of each \( R_i.P \) is performed based on the local knowledge base \( R_i.kb \) of robot \( R_i \). Thus, each transition \( t \in R_i.T \) is fired, if \( t \) is enabled and if \( R_i.kb \models t.\phi \). The distributed evolution of a set of PNPs \( \{ R_i.P = \{ R_i.P, R_i.T, R_i.F, R_i.W, M_0, G \} \} \) is a set of evolutions \( e_i = (\langle M_0, R_i.kb_0 \rangle, \ldots, \langle M_n, R_i.kb_n \rangle) \), one for each \( R_i.P \).

In order to compare the behavior of a centralized execution with respect to a distributed one, we introduce the concept of behavioral evolutions. Behavioral evolutions show the evolution of the behavior of the robots.

**Definition 9 (Behavioral Evolution)** Given the evolution \( e = (\langle M_0, R_i.kb_0 \rangle, \ldots, \langle M_n, R_i.kb_n \rangle) \), we say that its behavioral evolution is \( be = (\langle M_0^b, R_i.kb_0 \rangle, \ldots, \langle M_n^b, R_i.kb_n \rangle) \), where \( M_j^b \) is the projections of \( M_j \), obtained by ignoring the places of multi-robot operators, such as send, receive, hard and soft synchronizations.

We can prove that a centralized execution by Algorithm 1 of a PNP, in the case of multi-body planning, and its distributed execution by DEA, produce the same behavioral evolutions.

**Theorem 2** DEA correctly executes any PNP \( P \), i.e.: it produces the same behavioral evolution of the centralized execution of \( P \) with the same input.

**Proof** Sketch
During the execution of a multi-robot PNP, each robot executes the portions of the plan which are associated to it, except for $h_{\text{sync}}$ and $s_{\text{sync}}$ operators which require a central operator. The idea is that the DEA algorithm performs a distributed execution of the $h_{\text{sync}}$ and $s_{\text{sync}}$ operators. Clearly, if the distributed version of the synchronization operators has the same input-output behavior (considering that it cannot be interrupted), DEA produces the same behavioral evolution with respect to the centralized execution.

The proof can be obtained by showing that $h_{\text{sync}}$ and $s_{\text{sync}}$ have the same input-output behavior for both centralized and distributed execution. It can be shown that this property holds, by considering the behavior of the communication primitives $\text{send}(id, R_2)$ and $\text{receive}(id, R_1)$, used for the distributed model of the synchronization operators. In particular, to decompose a plan (as shown in Figures 10 and 9), we need to enumerate synchronization operators and associate to each of them a unique $id$. This $id$ will be used as the message sent by $\text{send}(id, R_2)$, and can be received only by a receive primitive with the same $id$. We then obtain the proof, if we consider the blocking or nonblocking behavior of the communication primitives during distributed execution. □

6 Cooperation using PNPs

In this section, we describe the use of PNPs to model cooperation in a team of robots. Cooperation is, in fact, a key feature in the design of a large number of multi-robot applications. PNPs can be used to achieve a simple form of dynamic task assignment by exploiting the coordination mechanisms. Nevertheless, coordination mechanisms are not enough to model more general forms of explicit cooperation. To this end, we introduce a synchronization mechanism, called multi-robot interrupt. Multi-robot interrupts, can be used to model explicit forms of cooperation such as Cohen and Levesque’s Joint Intentions theory [CL91, Tam97], also allowing for a distributed execution.

6.1 Task Assignment

An example of multi-robot cooperation is given by task assignment, which is the problem of assigning a set of tasks to a set of robots. Notice that, in multi-robot systems the knowledge bases $R_i.kb$ and $R_j.kb$ of two different robots, due to perceptual errors, may be inconsistent. Thus, it may be impossible to agree on a common description of the current situation. There are a number of papers which explicitly address task assignment problems for robots (e.g., Token Passing [FINZ06], Market Based [DS02, ZSDT02], Reactive Task Assignment [INPS03, WM00], Iterative Task Assignment [Par98] or Sequential Task Assignment [GM00, DS01, CCK92]). Among them, we consider the PNP implementation of a dynamic task assignment approach based on utility functions [INPS03], which has been demonstrated to be effective on real robots, especially when facing a dynamic environment. The main idea is that each robot, based on its local knowledge, broadcasts to its teammates its utility in performing each role. Then, each robots decides, based on the common knowledge of the utilities, which task it must perform.

Example 3 Consider a task assignment problem with two robots and two roles: two soccer robots need to perform a pass. Each of the two robots needs to initially decide whether it should pass or receive the ball. The robot that is closest to the ball is typically required to execute the pass task, whereas the robot which is more distant from the ball should execute
the \textit{receive} task. A PNP that models the required task assignment is shown in Figure 11. Once the soccer ball has been successfully located by the two robots (the portion of plan for finding the location of the ball is not shown in the figure), a $h\_sync$ operator is used to synchronize the execution and to exchange information about each robot’s utility in each of the two roles. This communication ensures that both robots share the same set of beliefs about their individual utilities. The task assignment is then consistently performed, in this case through the evaluation of the condition \textit{closestToBall}. In case Robot\textsubscript{1} is the closest to the ball ($R1\textunderscore closestToBall \text{ is true}$), the robot will perform a pass task. The pass and receive procedures are encoded in the remaining branches of the plan, not shown in the figure. The example considered the case of two robots and two roles, but the task assignment can be extended to the case of a larger number of robots and roles.

6.2 Multi-Robot Interrupt

A common problem, in multi-robot systems, is the ability to react quickly to changes in the environment. For example, consider again the problem of two soccer robots, one which was assigned the task to pass the ball, and one which was assigned the task to receive it. Both robots are committed to the coordinated task, and thus one robot will wait for a pass until the other robot will perform the pass. What if an opponent robot steals the ball, and the passer fails his task? If the passer robot perceives the event, he should abort the passing behavior and move on to some recovery strategy. Nevertheless, also the passer robot should
be notified, or he would wait for the pass forever. A possible solution is to use \_sync, as shown in Figure 12.

![Multi-robot interrupt modeled with \_sync.](image)

The problem with this solution is that a deadlock could be generated, since the condition "lostBall" potentially does not interrupt both robots' activities as desired. Consider the case that R2.receiveBall terminates before R1.pass ends. This may happen for several reasons, such as wrong perceptions or timeouts for behaviors. If at this stage R1 is still executing R1.pass and R1.lostBall becomes true, the execution of the pass action will be interrupted. R1 would prepare for a synchronization with R2, but this would never happen. R1 will indefinitely wait for R2's synchronization, resulting in an undesired deadlock. If only relying on hard sync operators it would be hard to avoid such situations during the design of a multi-robot plan.

![Multi-robot interrupt operator.](image)

**Multi-robot Interrupt Operator.** In order to implement the desired behavior, we introduce a synchronization mechanism, called *multi-robot interrupt* which is shown in Figure 13. Figure 14 shows the decomposition of the operator. The figures show the case for two robots, but it can be easily extended to the case of \( n \) robots. The multi-robot interrupt is similar to \_sync, but it connects two execution places of two different robots. The mechanism ensures that the interrupt occurs only if both the two robots are actually executing the two
Fig. 14 An example of the multi-robot interrupt operator decomposition: (a) multi-robot plan (b) single-robot plans

actions, thus avoiding the possibility of a deadlock highlighted in the previous example. The transition has a label showing the robot, and thus the \( k_b \), on which to evaluate the interrupt condition. The multi robot interrupt operator is characterized by \( T_C = \{t_i\} \), \( P_I = \{p_i_1, p_i_2\} \) and \( P_O = \{p_o_1, p_o_2\} \).

Formally, consider two robots \( R_1 \) and \( R_2 \), four PNPs \( R_1.P_1, R_1.P_2, R_2.P_1, \) and \( R_2.P_2 \), and the multi-robot interrupt operator. The PN formed by the interrupt of an action in \( R_1.P_1 \) through the place \( e_1 \), the interrupt of an action in \( R_2.P_1 \) and through the place \( e_2 \), the sequence of \( R_1.P_2 \) and the multi-robot interrupt through the place \( p_o_1 \), and the sequence of \( R_2.P_2 \) and the multi-robot interrupt through the place \( p_o_2 \) is a multi-robot PNP.

The semantics of distributed execution described in Section 5 is extended to the multi-robot interrupt. Single robot communication primitives are again used to communicate the
need for an action interruption among different robots. In particular, we use, a non-blocking
send primitive to notify other robots participating to the interrupt, and a blocking receive
primitive to fire the interrupt transition of the receiver robots. For the sake of readability, in
the following we represent the single-robot operators describing multi-robot interrupts with
a single transitions, used to represent the nets in Figure 14(b).

As a consequence of Theorem 2 and of the correspondence of the input-output behavior
of the centralized and the distributed multi-robot interrupt, it is easy to verify that DEA
correctly executes any multi-robot PNP $P$ which includes multi-robot interrupts.

6.3 Joint Intentions

Cooperation in multi-robot systems plays an important role, as teamwork can lead to consis-
tent performance improvements. Cohen and Levesque’s Joint Commitment theory provides
a detailed formal specification for the design of cooperative behaviors. Its prescriptive ap-
proach is easily expressed using PNs, which provide the required level of expressiveness,
while maintaining the desired generality to allow for the design of a wide range of coopera-
tive tasks. This section briefly summarizes the concepts behind the JC theory, and shows
how it is possible to use it for the design a PNP for explicit cooperation among multiple
robots.

The Joint Commitment theory isolates a set of basic requirements that all the cooperat-
ing members of a team should fulfill. The theory is rooted in the concept of commitment:
members that are committed to the execution of a cooperative behavior will continue their
individual action execution until one of the following conditions holds:

1. The behavior was concluded successfully
2. The behavior will never be concluded successfully (it is impossible)
3. The behavior became irrelevant

The prescriptive approach of the Joint Commitment (JC) theory can be used to provide a
systematic design of cooperative behaviors in a multi-robot team.

The concepts behind the JC theory are easily embodied in the design of multi-robot plans
for cooperative tasks. The multi-robot interrupt operator is used to consistently interrupt the
action execution among the different robots that are engaged in a cooperative behavior (be-
ing committed), in case the behavior became irrelevant or failed. The successful conclusion
of the individual actions is implemented in the multi-robot plan through a hard-sync oper-
ator. Figure 15 shows a multi-robot Petri Net Plan for a cooperative behavior, according to
the specifications of the JC theory.

After a first synchronization (during which the commitment is established), the two
robots perform the cooperative behavior, executing their individual actions (behavior1 and
behavior2). Following the JC theory, the commitment is broken, if one of the above listed
conditions holds. In case one of the engaged robots senses that the action became irrelevant
or that the action has failed, the multi-robot interrupts ensure that the event is communicated
to the partner, and the execution of the individual actions is interrupted. In case of successful
termination of both behavior1 and behavior2, a hard sync is performed to successfully end
the commitment. It may happen that one of the two robots successfully terminates the execu-
tion of the cooperative behavior, while the other is still performing some actions. To handle
this possibility and to prevent a deadlock situation to occur, the conditions for unsuccessful
commitment breaking have been duplicated.

In Figure 16, the single robot plan for robot $R2$ is shown. Only one of the two interrupt
Fig. 15 A Petri Net Plan for a cooperative behavior

Fig. 16 Single robot plan for the cooperative behavior
transitions connected to the execution of behavior2 can fire during the execution, as the other robot will only handle one of the two possible multi-robot interrupt messages.

7 Case Studies

The proposed framework has been implemented and used to control different robotic systems in different domains. A plan executor for PNP 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.

Among several applications realized with this framework, we describe here two experimental tests implemented with AIBO robots: a cooperative foraging test and a ball 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. In particular, we briefly sketch a case study on multi-robot foraging [FINZ06], which shows an example of complex coordination. Then, we detail a case study on robotic soccer, which presents many of the issues related both to coordination and cooperation. This last application was presented as a demo at AAMAS’08 conference and won the Best Robotic Demo Award [PZI+08]. Complete multi-robot plans, derived single robot plans and videos showing the execution of these tasks with the AIBO robots are available at pnp.dis.uniroma1.it.

7.1 Coordination in Robotic Foraging

In the first case study, we focus on coordination issues in a multi-robot foraging domain. Here we show the use of complex coordination for object manipulation, delegating cooperative issues to an external algorithm. While the coordination algorithm, as well as a detailed description of the experiments, can be found in [FINZ06], in this paper we describe the PNP s which have been used to coordinate the robots.

The multi-robot foraging test we have considered involves three robots that perform a synchronized operation on a set of similar objects scattered in the environment. In order to collect the objects, 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. An external module is used to dynamically assign tasks (a collector and a supporter) to the robots. The robots then execute a multi-robot PNP to jointly grab the objects and collect them in a predefined location in the environment.

![Diagram of multi-robot PNP for foraging test](image1.png)

**Fig. 18** Multi-robot PNP for foraging test.

Figure 18 shows the multi-robot plan used for this test. Hard synchronization is used to synchronize the robots 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, the collector robot grabs the ball and the supporter 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.

![Image of AIBO Robots during the passing task](image2.png)

**Fig. 19** AIBO Robots during the passing task.

### 7.2 Cooperation in Robotic Soccer

The second case study shows a complete example of using PNPs, including both coordination and cooperation issues. The goal of the case study is to have two robots passing the ball to each other. We require the resulting behavior to be robust to both action failures and exogenous events. To this end, we need to model, through the PNP, action synchronization,
dynamic task assignment and teamwork implemented through Joint Commitments theory. Thus, in contrast with the previous test, here there is no external module for cooperation. Indeed, cooperation is accomplished by using the PNP structures described in the previous sections.

The multi-robot plan is divided in three phases: 1) task assignment based on the distance to the ball, 2) preparation to the pass, 3) actual pass behavior.

In the task assignment phase, the robot which is close to the ball will take the role of the Passer and the other robot will behave as the Receiver. This is obtained by using the PNP task assignment mechanism illustrated by Figure 11. Note that this assignment is dynamic and depends on the actual position of the ball.

After the task assignment phase, the robots are committed to the execution of their tasks for passing/receiving the ball. The Passer robot moves to reach the ball, grabs it and rotates towards its partner. In the meantime, the Receiver robot reaches the desired position and prepares to intercept the passed ball, by rotating towards the Passer. At the end of this phase, the robots renew their commitment through another synchronization. The $h_{\text{sync}}$ operator is again used to ensure that both the robots have completed their actions, before they can proceed with the pass. This preparation phase is prone to action failures, due to the difficulty of implementing reliable grab and rotation primitives with AIBO robots, and due to possible occurrence of exogenous events (e.g. collisions with other robots), that may interfere with the predicted performance of the primitives. Reflecting the principles of the JC theory, the robots break their commitment in case a failure occurs during this phase (in this particular task the cooperative behavior is never considered irrelevant, as the robots have the unique task of passing the ball). More specifically, the LostBall condition becomes true, whenever the Passer robot realizes that the ball has been lost during the grab or the rotation phases. The ball may in fact roll away from the robot, causing the need for a new task assignment procedure. In case the control of the ball is lost by the Passer robot, the Receiver robot needs to be notified, in order to break its commitment to the current execution of the pass. A multi-robot interrupt operator is used to consistently interrupt the execution of the actions of both the Passer and the Receiver (Figure 20).

In case the preparation phase is successfully completed, the pass can take place. The Passer robot kicks the ball towards the Receiver, that in the meanwhile performs an intercept behavior. This phase does not require special attention for action interruption, as the kick and the intercept behaviors are atomically performed and the pass behavior is concluded both in case of success and in case of failure of the pass. A further synchronization (through a hard sync operator) is performed to exchange information about the outcome of the behavior, and the commitment is broken.

7.3 Example of PNP Validation

Figure 22 shows another possible experimental test which mixes the two previous ones: a collector robot collects balls as they arrive, when it is available, and then it passes the collected ball to a supporter robot, that stores it somewhere. In this richer setting, it is possible to identify several design features that can be supported by the analysis of PNP:

- the 2-robot system can recover from commitment loss when the collector C fails, but not when the supporter S fails. This means the net would not be reversible, though the subnet for the collector C would. This could be found by determining the underlying PN T-invariants;
Fig. 20 Preparation phase of the pass behavior

Fig. 21 Multi-robot Petri Net Plan for the pass behavior

– in fact, there is a deadlock when the reached marking has a single token in any of the places, after commitment break on the supporter S subnet, therefore the net is not live, and the goal of continuously collecting balls is not achievable (will be undermined by a commitment break by S);
– if we replace the current $h_{\text{sync}}$ with a $s_{\text{sync}}$, assuming the collector C will throw the ball to wherever the supporter S is, whenever C grabs a ball, another unbounded place would show up at the new $s_{\text{sync}}$ place, meaning that the collector performs its task faster than the supporter, an undesirable feature.
Fig. 22 A foraging task with pass behavior and commitment. The PNP in this example is not valid.
As this example shows, PN analysis tools can be used to study several features of PNP's. In general, these features may not only be limited to safeness, effectiveness and minimality as presented in Section 3.4. Further, discussion of analysis of PNP will be addressed as future work.

8 Conclusions

In this paper, we have presented a new formalism for high level programming of multi-robot systems that allows to represent plans with many important features such as sensing, loops, concurrency, 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. The high expressiveness of PNP allows for effectively capturing and dealing with most of the situations encountered when designing autonomous robots and multi-robot systems. By relying on PNs, 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.

In the future, we would like to address automatic plan generation, at least for subclasses of PNP's, by establishing a stronger link with reasoning about actions in dynamic systems.

References


