Auton Agent Multi-Agent Syst (2011) 23:344–383
DOI 10.1007/s10458-010-9146-1

Petri Net Plans

A framework for collaboration and coordination in multi-robot systems

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

Published online: 31 July 2010

Abstract Programming the behavior of multi-robot systems is a challenging task which has a key role in developing effective systems in many application domains. In this paper, we present Petri Net Plans (PNPs), a language based on Petri Nets (PNs), which allows for intuitive and effective robot and multi-robot behavior design. PNPs are very expressive and support a rich set of features that are critical to develop robotic applications, including sensing, interrupts and concurrency. As a central feature, PNPs allow for a formal analysis of plans based on standard PN tools. Moreover, PNPs are suitable for modeling multi-robot systems and the developed behaviors can be executed in a distributed setting, while preserving the properties of the modeled system. PNPs have been deployed in several robotic platforms in different application domains. In this paper, we report three case studies, which address complex single robot plans, coordination and collaboration.

Keywords Petri Nets · Multi-robot systems · Formal models · Plan representation and execution

V. A. Ziparo (✉) · L. Iocchi · D. Nardi · P. F. Palamara
Dipartimento di Informatica e Sistemistica "Antonio Ruberti" (DIS), Sapienza University of Rome, Via Ariosto 25, 00185 Roma, Italy
e-mail: ziparo@dis.uniroma1.it

L. Iocchi
e-mail: iocchi@dis.uniroma1.it

D. Nardi
e-mail: nardi@dis.uniroma1.it

P. F. Palamara
e-mail: palamara@dis.uniroma1.it

P. U. Lima
Institute for Systems and Robotics (ISR), Instituto Superior Técnico (IST), Lisbon, Portugal
e-mail: pal@isr.ist.utl.pt

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 collaboration among multiple robots, which must act as a team: teamwork can indeed boost performance.

Typically, complex multi-agent (or robot) behaviors, or more specifically plans, can be achieved through:

– *Plan design*: based on a representation formalism, an expert designs by hand the behaviors, which allow for the accomplishment of a given task;
– *Plan generation*: 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 rich 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.

In this article, we present a representation and execution framework for high level multi-robot plan design, called Petri Net Plans (PNPs) [58,59]. The goal of PNPs is to support developers in designing and implementing complex high-level robot and multi-robot behaviors, by providing a rich modeling language that offers several key features (such as, modeling concurrency, distributed execution, formal analysis, etc. …) that are very often required in robotic systems, but that have not been previously integrated into state of the art frameworks.

The syntax and the semantics of PNPs is based on Petri Nets (PNs) [40] and, indeed, PNPs are the first systematic and methodological approach for plan design based on PNs. PNPs inherit from PNs many of their features, which are very useful in robotic applications. An additional advantage of using PNs is that they have an appealing and intuitive graphical representation, that is (sometimes exponentially) more compact than state of the art behavior representation languages. Such graphical representation allows for both: at design time, understanding static properties of the net and, at runtime, visually monitoring the evolution of tokens in the net, and thus the actual robot behavior. The tools for task design, development and debug, based on "Petri Net languages" have been extensively used in distinct robotic applications by the many students involved in our projects. While a comprehensive and formal evaluation of how users work with the framework is out of the scope of this paper, our experience suggests that PNPs are more intuitive and easy to use as compared with competing approaches.

Despite the usability and intuitiveness of PNPs, several types of errors may frequently occur in plan design. Consequently, we identify some properties of plans that, when verified, prevent the designer from incurring into these errors. We show that verifying such properties can be reduced to standard PN analysis problems. These problems can then be solved using standard PN analysis tools, thus enabling for debugging tools to support plan design. The support on formal models is fundamental to ensure that plan formal specifications are satisfied, and represents a feature not often found in current robot task "models" and architectures.

It is worth noticing that, when using other formalisms, plan verification is typically based on empirical evaluation.

PNPs take inspiration from action languages (e.g., [45]), and, thus, are explicitly defined as composition of actions. 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 [10,47]. 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, its 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 perform complex tasks, which are not achievable by a single robot, and to avoid interference. To this end, we introduce coordination operators, which allow the designer to ensure synchronization constraints among actions of different robots. In a PNP, 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 a multi-robot PNP into a set of single-robot PNPs, which can thus 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.

PNPs can also be used for the implementation of *collaborative* behaviors. In particular, we show that in order to model collaborative behaviors, coordination is not enough. Thus, we introduce a new operator, the joint committed action, which we use to model a general theory of teamwork, namely the Joint Intentions theory. As for coordination, we show that collaborative behaviors described through PNPs allow for distributed execution.

The proposed framework has been implemented and is available[1] both as a $C++$ library and as an Open-RDK [2] 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 [3], Soccer [33,59], Foraging [16] and Manufacturing [26]). A PNP implementation for the soccer domain obtained the Best Robotic Demo Award at AAMAS'08 [41]. In this article, we present three case studies which show many of the relevant features of PNPs: a single robot task for search and rescue, complex coordination in a collaborative foraging problem and advanced collaboration using the Joint Intentions theory in a soccer domain. The case studies have been implemented on the AIBO robots and on a Pioneer wheeled platform.

In summary, the contributions of the proposed approach are manifold:

1. a methodological approach to Plan Design through PNs that, as a central feature, allows for automated plan verification;
2. a formal model for coordination and collaboration in multi-robot systems;
3. 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 PNPs in Sect. 3. Then, in Sect. 4 we provide operational semantics in terms of an execution algorithm for PNPs. In Sect. 5 we show some advanced features of the language for coordination, which include operators for synchronizing actions of different

[1] Available at pnp.dis.uniroma1.it.

robots and a distributed execution algorithm. Finally, in Sect. 6, we show how PNPs can be used to achieve collaboration. We conclude by showing some of the multi-robot systems we have implemented in Sect. 7 and by discussing the features of the proposed approach in Sect. 8.

2 Related work

This section gives an overview of the main approaches that have been proposed in the past few years for the representation and execution of robotic behaviors, in order to provide the context in which PNPs have been developed. We identify three broad classes of approaches to plan design and high-level programming for intelligent robots: *FSA-based approaches*, *BDI approaches* and *PN-based approaches*. We conclude the section with a comparative discussion of the these approaches with respect to PNPs.

It is worth noticing that there are a number of approaches that do not fall in the aforementioned three categories. These approaches are mostly programming tools (frameworks) for robotics, with no underlying formal model or with a limited underlying formal model, that usually have ad-hoc semantics and do not support formal analysis, making it difficult to develop robust and effective behaviors. The resulting tools often take the form of frameworks for ordinary programming languages. For example, ESL [20] is a language based on LISP that defines several constructs commonly used in robotics. In a similar way, the Task Description Language (TDL) [49] 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. The Reactive Action Packages (RAPs) [19] are expressed in LISP-like syntax and describe concurrent tasks along with execution constrains. RAPs are an ad-hoc tool for execution of concurrent tasks in robotic applications that have some similarities with PNs. In ESL, TDL and RAP no analysis of the resulting behavior is possible and coding coherent behaviors requires a considerable modeling effort.

2.1 FSA-based approaches

Many robot programming languages 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. FSA-based approaches stem from the need to implement effective behaviors in real-time systems [20]. Several frameworks have been implemented, proving their effectiveness in real world applications (e.g., [21]). 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. Notably, some methods for an automated FSA-based plan generation have also been developed (e.g. [34]).

Colbert [31] is a robot programming language that was developed as a component of the Saphira architecture [32]. Colbert has a syntax which is a subset of ANSI C, while 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 even though, in this case, the semantics are considerably different from standard FSA semantics and it is very hard to ensure coherence in the behaviors.

Xabsl [36] is a more recent approach and is based on hierarchical finite state automata. Xabsl is bundled with a set of language specific tools, which allow for an efficient development of behaviors.

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.

2.2 Belief, Desire, Intention

The Belief, Desire, Intention framework (BDI) [44] has been proposed as an alternative to FSA-based robot programming. In a BDI architecture, an agent selects behaviors to be executed (intentions), based on its goals (desires), and the current representation of the environment's state (beliefs). The procedural knowledge is typically encoded in a predefined library of plans (e.g. [22]). In order to obtain the desired balance between reactivity and goal-directed behaviors, an agent can commit to the execution of plans and periodically reconsider them. One key advantage of BDI over FSA-based robot programming is that the designer needs not specify a predefined ordering of basic behaviors, allowing him or her to draw the executed plans from a potentially very large search space. On the other hand, no automated planners nor validation tools are available for BDI frameworks (though see [12,50] for recent promising developments).

Several architectures inspired by the BDI framework have been proposed for modeling Multi-Agent Systems (MAS). Two notable such architectures that also model collaboration among multiple agents are STEAM [51] and, more recently, BITE [28].

STEAM is implemented with a focus on collaborative behaviors, by relying on Cohen and Levesque's *Joint Intentions* Theory [6]. Agents in a STEAM architecture distributedly monitor the execution of collaborative behaviors (which are organized in a partial hierarchy of joint intentions), possibly reorganizing the team. Cooperation is implemented in STEAM through a set of complex domain-independent rules, incorporated in the architecture to form sophisticated hierarchical team structures. STEAM was used in a number of applications (e.g. simulated military missions and virtual soccer players), but never on real robotic platforms.

The BITE architecture was specifically designed for robotic applications that involve collaboration and coordination. To manage teamwork, BITE maintains an organization hierarchy, a task/sub-task behavior graph, and a library of hierarchically linked social interaction behaviors. Although no explicit methodological guidance to teamwork design is provided, one of the strengths of the BITE architecture is the possibility to specify different types of interaction templates (e.g. for synchronization and task allocation), which can be reused to automate individual robots' task selection in different scenarios. BITE is focused on the automation of teamwork, while the design of individual behaviors is not extensively addressed. As with other systems based on BDI, BITE does not provide formal validation tools to verify the consistency of the designed behaviors.

The general BDI approach is orthogonal to PNPs, as PNPs can be used as a representation formalism for intentions. In fact, in Sect. 6.2, we show an implementation of a joint action, following the guidelines of the *Joint Intentions* Theory used in STEAM. Moreover, PNPs have an organization hierarchy (plans/sub-plans) as in BITE, but are not limited only to model teamwork. Finally, the use of PNPs allows for exploiting standard validation tools based on PNs to verify the consistency of the designed behaviors.

2.3 PN-based approaches

A solution to robot programming that recently gained interest in the scientific community is the modeling of robotic behaviors through PNs. In particular, there has been a considerable effort in modeling MAS through PNs [17], given their capability of representing concurrent systems and shared resources. PN-based systems offer two main advantages with respect to FSA [40]: PN languages (languages marked by PNs) are a super-set of regular languages (languages marked by FSA), due to memory and concurrency characteristics. Therefore, the set of modeled roles and behaviors is potentially richer when using PNs; secondly, PNs allow for automatic analysis and verification of formal properties on the performance of the modeled systems. Available tools such as PIPE [1] or TimeNET [57] check PN properties, both through simulation and closed-form equivalent Markov Chain analysis.[2]

Therefore, PNs have been widely used in the literature to model Discrete Event Systems (DES), namely manufacturing systems [53]. PN-based models of robot tasks started with the pioneer work of Wang et al. [54], where they were used to implement the Coordination Level of Saridis' 3-level hierarchy for intelligent machines, including reinforcement learning algorithms. In the past few years, approaches to plan generation and representation based on PNs, have gained increasing interest, addressing both multi-agent (MAS) and multi-robot systems (MRS).

Action representation using PNs in MAS is proposed, for example, in [4]. The model is limited to purely reactive agents: actions are instantaneous, as they are represented by PN transitions, and the places of the PN model represent the environmental state of the agent. In a MAS framework, issues that are typical of embodied agents such as non-instantaneous actions or uncertain action effects are not specifically addressed. Interactions among agents have also been modeled by PNs in the literature, with a special focus on the formal modeling of conversations using colored PNs [8]. A substantial comparative review of the different approaches, including a colored PN model of multi-agent conversations, where places explicitly represent joint interaction states and messages, can be found in [25]. Poutakidis et al. [43] introduced interaction protocols, specified using Agents UML and translated to PNs, to debug agent interaction. The debugger uses the PNs to monitor conversations and to detect when protocols are not correctly followed by the agents. While these works provide formal scalable models of interaction, they do not model actions explicitly, and they are not concerned with commitment issues, which are relevant in applications where the interaction among agents/robots is cooperative.

A first group of works using PNs for designing robotic systems develops ad-hoc models for specific applications rather than providing a formal language for robot programming. For example, in [48], PNs are used to model a multi-robot coordination algorithm which is based on an auction mechanism to perform environment exploration. Similarly, [56] shows an agent based extension of Fuzzy Timed Object-Oriented PNs (proposed in [38]) for the design of collaborative multi-robot systems for a specific industrial application. Another example is [35], where the authors report the use of distributed agent-oriented PNs for the modeling of a Multi-Robot System for playing soccer. These works focus mainly on the PN that controls the robotic system and on its execution, providing no systematic method to program such a controller using PNs and/or for the analysis of the whole system properties.

[2] For an exhaustive list of available tools see http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/.

A second kind of PN-based approaches models the robotic system, representing plans as PNs so as to analyze their properties and/or to synthesize optimal plans from conditional ones. In [37], the authors propose an approach for modeling single-robot systems. In this case, users define several possible plans to carry out 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-robot approach has been tested in two real world applications regarding manipulators and mobile robots. A follow-up work is presented in [9], but in this case the PN model explicitly includes a representation of the environment. Nevertheless, none of these works provide a formal PN-based language for plan description and composition. Furthermore, though they focus on modeling, no models of cooperation or coordination are proposed.

A third category of works addresses specification and execution monitoring of plans for multi-robot systems using PNs. The compilation of plans for multiple robots into PNs for analysis, execution, and monitoring is proposed by King et al. [29]. In this work, plans for each single robot are generated either by using a graphical interface or by using some automated planning method. The operators that are used for the PN representation of the plans are inspired by the STRIPS [18] planning system. Supervisory control techniques are applied to the PN 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, re-planning is used at run-time, which severely limits the applicability of this approach to real-rime systems in dynamic environments. Novel supervisory control techniques are also introduced and applied to simulated and real sensor networks, thereby mixing static and mobile sensors in [24]. Another formal framework for robotic collaboration based on an extension to PNs, known as workflow nets, is introduced by Kotb et al. [30] to establish a protocol among mobile agents/robots based on the task coverage they maintain. PNs are used to ensure the soundness of the framework and to quantify task performance and determine goal state reachability. However, none of these works provides a formal PN-based language for plan description and composition.

Despite the large body of work on modeling robot and multi-robot behaviors through PNs, there currently is not a standard representation formalism for representing multi-robot systems based on PNs. Indeed, most approaches provide ad hoc solutions to specific problems.

2.4 Comparison with PNPs

Our goal is to provide a systematic approach to robot behavior programming and verification, which addresses cooperation in multi-robot teams supported by a well-defined plan specification language. Furthermore, the design methodology we present can exploit existing techniques and tools for plan analysis. To this end, we define PNPs (first introduced in [58]) as a subset of PNs, by relying on modeling primitives of action languages inspired by Situation Calculus [39], such as ConGolog [11], and specific structures to model cooperation among multiple robots, such as Joint Intentions Theory [6]. The resulting language is more expressive than most approaches in the literature, allowing for complex forms of sensing, loops, interrupts, concurrency, coordination and cooperation.

It is important to notice that PNPs are very expressive, but also allow for a compact representation of the behavior of a robot. In comparison with other well-known models for transition systems based on FSA, PNs (and thus PNPs) are in general exponentially more compact. For example, a finite structure (but unbounded) PN can represent even an infinite state automaton. Specifically, we stress that PNPs are more powerful when compared to FSA

in modeling concurrency. In fact, in FSA-based representations each state of the system is associated with a node of the automaton, and the execution of concurrent actions requires to define many states according to all the possible combinations of events occurring during the parallel execution. The number of FSA states needed to represent the concurrent execution of different sequences of actions is thus exponential in the number of such sequences, while in PNPs the number of places and transitions is proportional to the number of these sequences.

BDI architectures try to address the issue of reducing the dimensionality of the space of plans by avoiding to commit to assigning a specific ordering in behaviors. This is indeed a very powerful approach in many application domains, but has the drawback that it is very hard to design plans, when the domain requires complex behaviors where atomic actions are highly coupled and the degree of optimality of the solution is a key factor.

PNs, like as FSA, implicitly commit to assigning a specific ordering of behaviors. Indeed, this is claimed as an advantage, because this way one reduces the dimensionality of the space of plans where the optimal plan must be searched. This stems from the claim that one can design simple FSA or PNs for components of the involved behaviors and their interaction with the world: the actual behavior results from the composition of those behaviors, filtering out combinations that can never occur and/or that the designer knows right from the start how to avoid. In fact, the initial components force some behavior sequences based on the designer expertise, which leaves the possibility of other combinations so as to enable different alternative plans for the same task (e.g., through conflicts in PNs). Clearly, quality and complexity of the plans comes at a cost: the modeling effort required for designing behaviors based on PNs.

In order to address the complexity of plan design, and opposed to state of the art PN-based approaches, PNPs provide a clear systematic methodology for modeling complex single robot and collaborative behaviors through PNs. PNPs are very intuitive and require a small modeling effort, due to the explicit characterization of atomic structures, which are explicitly interpreted as *actions* and *operators* to combine actions. PNPs support plan design through the use of PN analysis methods. These methods can be used to verify important properties of the plans which are required for a robust execution. Moreover, PNPs allow for a distributed execution of plans which has the notable characteristic of preserving the properties verified for the plans at design time.

3 Petri Net Plans syntax

Petri Net Plans 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. Initially we ignore multi-robot operators that are used to achieve coordination and collaboration. Thus, we can consider the multi-robot system, as composed by robots which execute their own PNP independently from each other. Nevertheless, as we will see in Sects. 5 and 6, PNPs naturally model multi-robot systems when enriched with multi-robot operators.

In the following, we first provide an interpretation of PNs for behavior execution, namely PNP structures. Then, we define the PNP Language (PNPL). PNPL is a high-level robot programming language that builds upon PNP structures and that provides a methodology for building PNP structures. Finally, we define PNPs as elements of PNPL, that must obey to runtime constraints. These constraints can be verified based on standard PN analysis tools. More details on the (operational)semantics of PNPs are given in Sect. 4.

3.1 PNP structures

PNP structures are PNs with a domain specific interpretation aimed at modeling robotic behaviors. In particular, they are PNs that have at most one token per place and edges of weight one.

Definition 1 (*PNP structure*) A PNP structure is a PN $pn = \langle P, T, F, W, M_0 \rangle$ and a goal marking G which specifies the set of desired termination states. We define a PNP structure as the following 6-tuple:

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

- $P = \{p_1, p_2, \ldots, p_m\}$ 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 structure.
- $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$
- $G : P \rightarrow \{0, 1\}$ is the goal marking.

PNP structures can be considered as PNs, with a domain specific interpretation and an extended semantics. In a PNP, places have different interpretations, thus they are partitioned into four classes: $P = P^{\mathcal{I}} \cup P^{\mathcal{O}} \cup P^{\mathcal{E}} \cup P^{\mathcal{C}}$, where:

1. $P^{\mathcal{I}}$ is the set of *input places*, which model initial configurations of the PNP;
2. $P^{\mathcal{O}}$ is the set of *output places*, which model final configurations of the PNP;
3. $P^{\mathcal{E}}$ is the set of *execution places*, which model the execution state of actions in the PNP;
4. $P^{\mathcal{C}}$ is the set of *connector places*, which are used to connect different PNPs.

Also transitions are partitioned in three subsets $T = T^{\mathcal{S}} \cup T^{\mathcal{T}} \cup T^{\mathcal{C}}$, where:

1. $T^{\mathcal{S}}$ is the set of start transitions, which model the beginning of an action/behavior;
2. $T^{\mathcal{T}}$ is the set of termination transitions, which model the termination of an action/behavior;
3. $T^{\mathcal{C}}$ is the set of control transitions, which are part of the definition of an operator.

A PNP structure 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 $e \in P^{\mathcal{E}}$ and $M(e)$ determines whether the behavior is active or not. Thus, the set of execution places of a PNP structure (i.e., $P^{\mathcal{E}}$) models the execution state of the system and its temporal evolution characterizes the system dynamics.

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

3.2 PNP Language

PNP Language defines a subset of PNP structures aimed at providing a methodology for designing PNP structures. We define the PNPL in terms of PNP structures, which can be build upon atomic *actions* and combined through *operators*. In the following, we will refer to PNP structures, simply as structures.

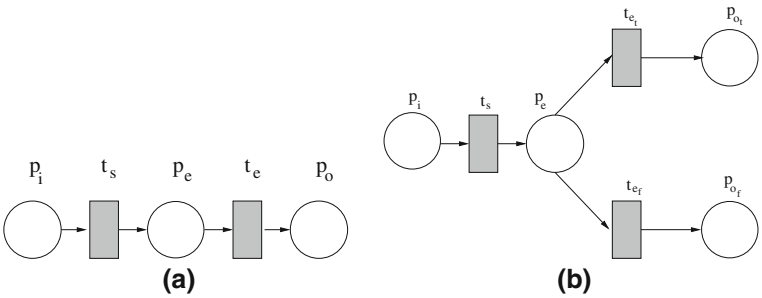

Fig. 1 Actions: **a** ordinary action and **b**sensing action

3.2.1 Actions

Actions represent primitive behaviors of robots and are the atomic concept upon which we build complex PNP structures. Actions are characterized by having $T^{\mathcal{C}} = \emptyset$. 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 the structure shown in Fig. 1a, where:

- $P^{\mathcal{I}} = \{p_i\}$, $P^{\mathcal{E}} = \{p_e\}$ and $P^{\mathcal{O}} = \{p_o\}$,
- $T^{\mathcal{S}} = \{t_s\}$ and $T^{\mathcal{T}} = \{t_e\}$.

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 the structure shown in Fig. 1b, where:

- $P^{\mathcal{I}} = \{p_i\}$, $P^{\mathcal{E}} = \{p_e\}$, $P^{\mathcal{O}} = \{p_{o_t}, p_{o_f}\}$,
- $T^{\mathcal{S}} = \{t_s\}$ and $T^{\mathcal{T}} = \{t_{e_t}, t_{e_f}\}$.

and where t_{e_t} and t_{e_f} are the transitions ending the action, when the sensed property is true, and when it is false, respectively. Analogously, the places p_{o_t} and p_{o_f} terminate the action, when the sensed property is true and when it is false. Notice that, it is possible to extend sensing actions in order to have more than two mutually exclusive outcomes by augmenting the number of termination transitions and output places. We also 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.

3.2.2 Operators

PNP structures are modular, since they allow for combining multiple structures in order to build more complex ones. Two structures can be combined by merging two places, one for each structure. This allows for sequencing behaviors and constructing loop structures. Moreover, it is possible to monitor the execution state of structures by using interrupt operators, that tie the execution places of an action with the input places of another structure, through a transition (interrupt) that suspends the execution of the current action and triggers

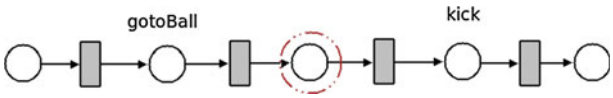

Fig. 2 The sequence of two ordinary actions

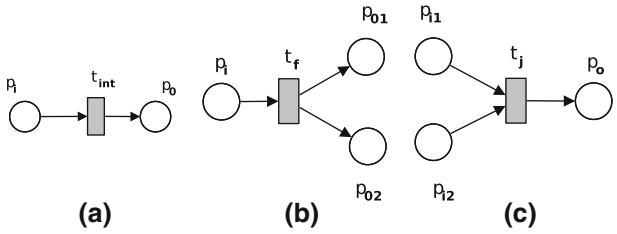

Fig. 3 Three operators: **a** Interrupt; **b** Fork; **c** Join

a recovery behavior. 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 single robot, or multi-robot actions.

Therefore, in order to create complex PNPs, four kinds of operators are defined: *sequence*, *interrupt*, *fork* and *join*. Operators are PNs used as control structures, which do not refer to specific behaviors. Operators are thus characterized by having $P^{\mathcal{E}} = \emptyset$, $T^{\mathcal{S}} = \emptyset$, and $T^{\mathcal{T}} = \emptyset$.

Sequence operator. The sequence operator combines two structures by merging two of their places. For example, an output place of a first structure can be merged with an input place of a second one, to obtain a chain of the two structures. The sequence of a motion behavior for approaching a ball (i.e., gotoball) and a kicking behavior are shown in Fig. 2. The dashed circle shows the result of merging 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 $PN_2 = \langle P_2, T_2, F_2 \rangle$, a non-execution place $p \in P_1^{\mathcal{I}} \cup P_1^{\mathcal{O}} \cup P_1^{\mathcal{C}}$ and an output place $o \in P_2^{\mathcal{O}}$, 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 $P = P_1 \cup P_2 - \{o\}$, $T = T_1 \cup T_2$, $F = F_1 \cup F_2 - \{\langle t_e, o \rangle\} \cup \bigcup_{t_e \in {}^\bullet o} \{\langle t_e, p \rangle\}$, where ${}^\bullet o$ is the set of input transitions of o. Moreover, $P^{\mathcal{I}} = P_1^{\mathcal{I}} \cup P_2^{\mathcal{I}} - \{p\}$, $P^{\mathcal{O}} = P_1^{\mathcal{O}} \cup P_2^{\mathcal{O}} - \{o\}$, $P^{\mathcal{E}} = P_1^{\mathcal{E}} \cup P_2^{\mathcal{E}}$, $P^{\mathcal{C}} = P_1^{\mathcal{C}} \cup P_2^{\mathcal{C}} \cup \{p\}$, if $p \in P_1^{\mathcal{I}}$, while $P^{\mathcal{C}} = P_1^{\mathcal{C}} \cup P_2^{\mathcal{C}}$ otherwise. This operator can be applied to two places of the same structure for creating loops. Thus, when $PN_2 = PN_1$, we call this operator *loop-sequence*.

Interrupt operator. The *interrupt* operator, shown in Fig. 3a, 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 Fig. 4, the gotoball action is monitored by an interrupt triggered when the ball is stolen by an opponent. As a recovery procedure, the robot starts a defensive behavior. The dashed circles show the result of merging the execution place of the gotoball action with the interrupt and the result of merging of the interrupt and the input place of the defend action. Interrupts can also be used to interrupt multiple actions simultaneously (see Fig. 6b for an example). Formally, given two PNP structures $PN_1 = \langle P_1, T_1, F_1 \rangle$ and $PN_2 = \langle P_2, T_2, F_2 \rangle$, a set of execution places $\{e_i\}$ such that $e_i \in P_1^{\mathcal{E}}$ and a non-execution place $p \in P_2^{\mathcal{I}} \cup P_2^{\mathcal{O}} \cup P_2^{\mathcal{C}}$, the *interrupt* of PN_1 through PN_2 is a PNP structure $PN = \langle P, T, F \rangle$, with $P = P_1 \cup P_2$, $T = T_1 \cup T_2 \cup \{t_{\text{int}}\}$,

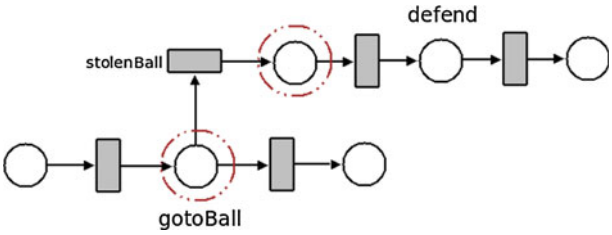

Fig. 4 The interrupt of an action and its recovery procedure

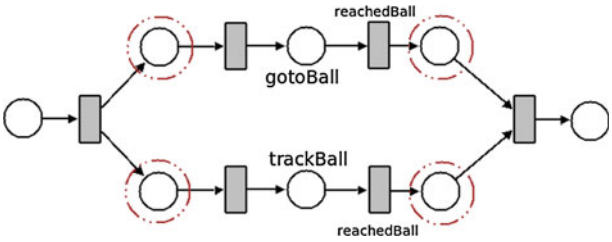

Fig. 5 The fork and the subsequent join of two actions

$F = F_1 \cup F_2 \cup \bigcup_i \{\langle e_i, t_{\text{int}}\rangle\} \cup \{\langle t_{\text{int}}, p\rangle\}$, where t_{int} is the transition associated to the interrupt condition. Moreover, $P^{\mathcal{I}} = P_1^{\mathcal{I}} \cup P_2^{\mathcal{I}} - \{p\}$, $P^{\mathcal{O}} = P_1^{\mathcal{O}} \cup P_2^{\mathcal{O}}$, $P^{\mathcal{E}} = P_1^{\mathcal{E}} \cup P_2^{\mathcal{E}}$, $P^{\mathcal{C}} = P_1^{\mathcal{C}} \cup P_2^{\mathcal{C}} \cup \{p\}$, if $p \in P_1^{\mathcal{I}}$, while $P^{\mathcal{C}} = P_1^{\mathcal{C}} \cup P_2^{\mathcal{C}}$ otherwise. Finally, $T^{\mathcal{C}} = T_1^{\mathcal{C}} \cup \{t_{\text{int}}\}$. 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, $P_2 = P_1$ and we call this operator *loop-interrupt*.

Many robotic systems are required to handle concurrency due to: (1) the possibility of actuating simultaneously, and independently, several parts of the body and (2) the possibility of controlling 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 Sects. 5 and 6. In particular, here we present fork and join operators for dealing with multiple actuators.

Fork operator. Each token in a structure can be thought as a thread of execution. The fork operator generates multiple threads from a single thread of execution. Figure 3b shows a fork structure producing two threads of execution. The fork operator is characterized by $T^{\mathcal{C}} = \{t_f\}$, $P^{\mathcal{I}} = \{p_i\}$ and $P^{\mathcal{O}} = \{p_{o1}, p_{o2}\}$. Notice that the operator can be extended to generate more threads by adding new output places. Formally, the fork operator of two PNP structures P_1 and P_2 is the *sequence* of a *fork* structure with P_1, through p_{o1}, and P_2, through p_{o2}. The left side of Fig. 5 shows the fork of the actions $gotoBall$ and $trackBall$. The dashed circles show the result of merging the output places of the fork operator with the input places of the actions.

Join operator. The join operator allows the synchronization of multiple threads of execution. This operator consumes multiple threads of execution simultaneously, and generates a single synchronized thread. The join structure is shown in Fig. 3c, for the case of two threads. As in the previous case, the operator can be generalized to synchronize more threads by adding

new input places. The join operator is characterized by $T^{\mathcal{C}} = \{t_j\}$, $P^{\mathcal{I}} = \{p_{i1}, p_{i2}\}$ and $P^{\mathcal{O}} = \{p_o\}$. Formally, the join operator of two PNP structures P_1 and P_2 is the *sequence* of P_1, through p_{i1}, and of P_2, through p_{i2}, with the *join* structure. The right side of Fig. 5 shows the join of the actions $gotoBall$ and $trackBall$. The dashed circles show the result of merging the output places of the actions with the input places of the join operator.

Notice that the fork and join operators allows for duplicating tokens, thus enabling concurrent execution of actions. Indeed, in the example before, when the execution places of both the actions $gotoBall$ and $trackBall$ are marked, the actions are actually executed in parallel. This mechanism is not limited to concurrent execution of two actions, but it can be used to model concurrent execution of complex behaviors. By using a different token for each execution thread, it is possible to model in a compact way all the possible combinations of event occurring during such a parallel execution. For example, the parallel execution of n sequences of actions of length k, is represented in a PNP with $O(k\,n)$ places and transitions, while it would require for example $O(k^n)$ states if using a FSA-based representation.

The PNPL describes the subset of PNP structures inductively defined as the closure of actions, under the *sequence*, *interrupt*, *fork* and *join* operators. In particular, the PNPL is the set of PNP structures described as follows.

Definition 2 (*PNP Language*) A PNP structure s is in PNPL (i.e., $s \in$ PNPL) if and only if there exist $s_1 \in$ PNPL and $s_2 \in$ PNPL, such that at least one of the following assertions hold:

- s is an ordinary or a sensing *action*;
- s is the *sequence* of s_1 and s_2;
- s is the *interrupt* of s_1 and s_2;
- s is the *fork* of s_1 and s_2;
- s is the *join* of s_1 and s_2.

3.3 PNP definition

In order to define behaviors that are actually executable, PNPs must fulfill some additional requirements on their behavior at runtime.

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 3 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 a 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, it can be fired at least once in some sequence of markings. This property corresponds to $L1$-liveness in PNs.

Definition 4 A PNP structure $\langle P, T, F, W, M_0, G\rangle$ is *minimal* if it is $L1$-live.

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

Finally, notice that, as a structural difference from 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 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 5 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 home 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 6 (*Petri Net Plan*) A Petri Net Plan (PNP) $\mathcal{P}$ is a PNP structure such that $\mathcal{P} \in$ PNPL and such that:

- $\mathcal{P}$ is safe (Definition 3)
- $\mathcal{P}$ is minimal (Definition 4)
- $\mathcal{P}$ is effective (Definition 5)

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

3.4 Sub-plans

In the design of a PNP, sub-plans can be used for modularity and readability. A sub-plan is represented as an ordinary action, but it refers to a structure 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 PNPs with sub-plans is outside of the scope of this paper. Nevertheless, the discussion which follows can be generalized in many cases to PNPs 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 PNPs 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.

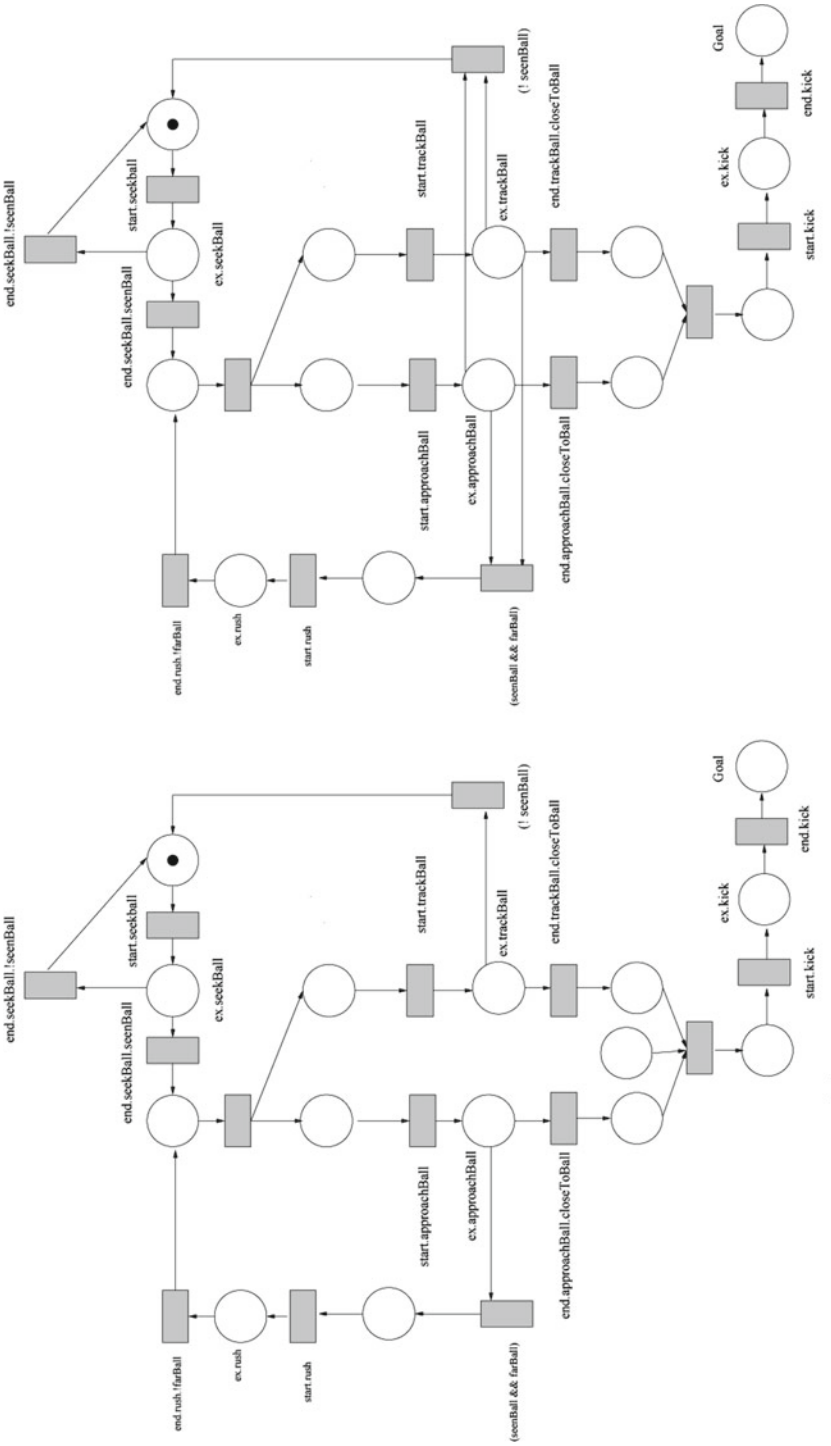

Fig. 6 A robotic soccer example: striker robot. The labeling shown is an example of the actual syntax used for our executable plans. Labels characterize action nodes and conditions associated to transitions. The net **a** denotes a PNP structure which is not a valid PNP, while net **b** is a valid PNP

The PNP in Fig. 6a 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 goal.

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 it is closer to the ball, the approach ball and the track ball behaviors 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 Fig. 6b.

4 Petri Net Plans execution

The operational semantics of PNPs, 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 PNs, is in the way PNPs 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 PNPs are the ones which correspond to action starts. In this paper, we assume that plans do not have non-deterministic choices for controllable transitions. 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 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 ϕ 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

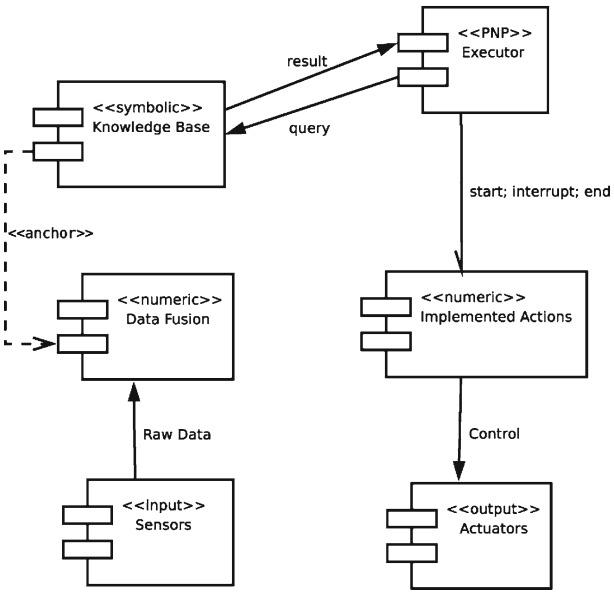

Fig. 7 An abstract robot architecture for PNPs

markings and knowledge bases $\langle M_i, kb_i \rangle$, representing the execution state and the knowledge of a robot at time τ_i.

Definition 7 (*Evolution*) An *Evolution* of a PNP $\mathcal{P} = \langle P, T, F, W, M_0, G \rangle$ is a temporally annotated sequence of pairs $(\langle M_0, kb_0 \rangle, \ldots, \langle M_n, kb_n \rangle)$. In particular, each $\langle M_i, kb_i \rangle$ represents a marking M_i obtained, given a knowledge base kb_i, at time τ_i.

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

Definition 8 (*Admissible Evolution*) An *Admissible Evolution* of a PNP $\mathcal{P} = \langle P, T, F, W, M_0, G \rangle$ is a an evolution $(\langle M_0, kb_0 \rangle, \ldots, \langle M_n, kb_n \rangle)$, where M_0 is the initial marking, $\forall_{i<n}\ M_i \notin G$ and such that:

$$\forall_{i \in \{0..n-1\}}\ \exists_{t \in T} \mid enabled(t, M_i) \ \wedge \ fire(M_i, t) = M_{i+1} \ \wedge \ kb_{i+1} \models t.\phi$$

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 Fig. 7. Nevertheless, the use of PNPs is not restricted to this particular type of architecture. Specifically, 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 formalism: for example, in our implementation, we use a simple conjunction of propositions. Analogously, queries 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 robot is able to evaluate queries over the current model of the world, i.e., to compute $kb \models t.\phi$.

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 [52], such as localization, mapping, tracking, etc. Notice that this numerical information must be anchored to the symbols in the knowledge base [7]. In order to effectively control the behavior of the robot, we assume the availability of a set of implemented actions $\mathcal{A} = \{a_1, \ldots, a_k\}$. 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. PNPs allow for both threaded and interleaved concurrency. In the threaded case, each action is executed in a separate thread with respect to other actions. This means that after an action is started, it will remain active until either *end* or *interrupt* is invoked. In the interleaved case, a unique thread is used both for the executor and all the actions. Roughly, the idea is that each action implements an $executeStep()$ method, which executes a single step of the action. The PNP executor then invokes at each cycle such methods for all actions which have a token in their execution place. Both execution models can be used for executing PNPs. For simplicity, in this paper we refer to the case of threaded execution.

4.2 PNP execution algorithm

Algorithm 1 is the execution algorithm for PNPs. As previously described, it relies on a Knowledge Base 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, 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 ($enabled(t, CurrentMarking)$) and if the related event occurs. If these two conditions are satisfied, the procedures for action control are handled within the sub-procedure $handleTransition$ (line 5) and the transition t is fired (line 6) resulting in a new marking. $handleTransition$ 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 $\mathcal{P}$, i.e.:*

- *any computed evolution $(\langle M_0, kb_0\rangle, \ldots, \langle M_n, kb_n\rangle)$ of $\mathcal{P}$ is an admissible evolution*
- *the behaviors of a robot are started, interrupted or ended, when the start, the end and interrupt transitions of the corresponding actions are fired;*

Algorithm 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** $enabled(t, CurrentMarking) \wedge kb \models t.\phi$ **then**
5: $handleTransition(t)$
6: $CurrentMarking = fire(CurrentMarking, t)$
7: **end if**
8: **end for**
9: **end while**

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**
$t.a.interrupt()$
end if

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 PNPs.

To prove that any evolution is admissible, we need to show that evolutions obey the firing rule (i.e. $enabled(t, M_i) \wedge fire(M_i, t) = M_{i+1}$), and that the conditions labeling transitions are correctly evaluated ($kb_{i+1} \models t.\phi$). The former requirement is explicitly satisfied by lines 4 and line 6, while the latter by line 4.

The second part of the proof is verified by a direct analysis of the procedure $handleTransition(\cdot)$. □

5 Coordination using PNPs

PNPs can be effectively used also to model multi-robot behaviors, allowing for dealing with the typical issues encountered when designing a multi-robot systems, in particular, action synchronization and joint intentions.

In the following, we show how PNPs can be used to support coordination so as 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 [15] for details). In our work, we follow *centralized planning*. Specifically, we provide a distributed execution model by implementing a *centralized planning for distributed plans* approach [15]. Our distributed execution model allows to execute a set of single-robot PNPs, derived from a multi-robot PNP, without the need of a central coordinator robot.

5.1 Synchronization operators

We can consider a multi-robot PNP, for a team of robots $R_1, \ldots, R_n$, 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 that 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 $\{\langle R_j.P, R_j.T, R_j.F\rangle\}_{j=1\ldots n}$ we define a multi-robot PNP as:

$$\mathcal{P} = \langle P, T, F\rangle$$

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

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 Fig. 8, supports time synchronization for the actions of two robots R_1 and R_2. The operator has two input places $P^{\mathcal{I}} = \{p_{i1}, p_{i2}\}$ and two output places $P^{\mathcal{O}} = \{p_{o1}, p_{o2}\}$ and $T^{\mathcal{C}} = \{t\}$. The operator is similar to a join and a fork, except that it is used to synchronize behaviors of different robots. Formally, consider two robots R_1 and R_2, four PNPs $R_1.\mathcal{P}_1$, $R_1.\mathcal{P}_2$, $R_2.\mathcal{P}_1$, and $R_2.\mathcal{P}_2$, and the *h_sync* operator. The PN formed by the *sequence* of $R_1.\mathcal{P}_1$ and *h_sync* through the place p_{i1}, the *sequence* of $R_2.\mathcal{P}_1$ and *h_sync* through the place p_{i2}, the *sequence* of $R_1.\mathcal{P}_2$ and *h_sync* through the place p_{o1}, and the *sequence* of $R_2.\mathcal{P}_2$ and *h_sync* through the place p_{o2} is a multi-robot PNP.

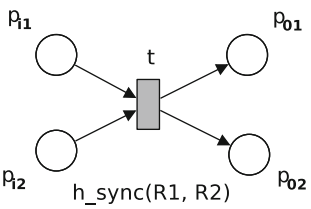

Fig. 8 A *h_sync* operator

Example 1 Figure 11a 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 h_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 $R_1.gotoLeftSideTable$ and $R_2.gotoRightSideTable$ 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 Fig. 9, can be used to force a precedence relation among the actions of two different robots. The operator has two input places $P^{\mathcal{I}} = \{p_{i1}, p_{i2}\}$, two output places $P^{\mathcal{O}} = \{p_{o1}, p_{o2}\}$, one connector place $P^{\mathcal{C}} = \{p_m\}$ and two control transitions $T^{\mathcal{C}} = \{t_f, t_j\}$. Formally, consider two robots R_1 and R_2, four PNPs $R_1.\mathcal{P}_1$, $R_1.\mathcal{P}_2$, $R_2.\mathcal{P}_1$, and $R_2.\mathcal{P}_2$, and the s_sync operator. The PN formed by the *sequence* of $R_1.\mathcal{P}_1$ and s_sync through the place p_{i1}, the *sequence* of $R_2.\mathcal{P}_1$ and s_sync through the place p_{i2}, the *sequence* of $R_1.\mathcal{P}_2$ and s_sync through the place p_{o1}, and the *sequence* of $R_2.\mathcal{P}_2$ and s_sync through the place p_{o2} is a multi-robot PNP.

Example 2 Figure 12a shows an example of the use of the s_sync. In this example, there are two robots R_1 and R_2. The first robot 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 s_sync. 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 as a single-robot PNP, in the case of multibody planning [46], 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 by providing 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 by 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 the behavior of synchronization operators in a distributed way, by replacing the synchronization operators with appropriate combinations of a non-blocking send action and a blocking receive action. The replacement of synchronization operators with communication actions leads to a distributed version of the firing rule, specific to synchronization structures.

We assume communication to be reliable. Moreover, we assume that the communication actions are instantaneous, and as such, we represent them as a single transition (see Fig. 10). The assumption of communication being instantaneous is often reasonable, given that robot actions involve moving actuators in the physical world that usually requires orders of magnitude more time than communication. So in this article we consider reliable communication

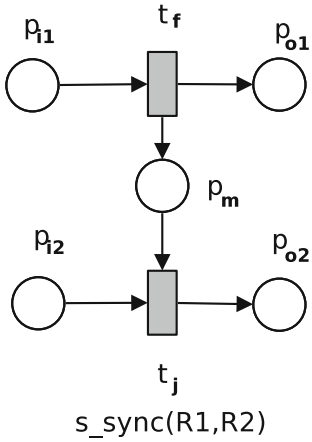

Fig. 9 A *s_sync* operator

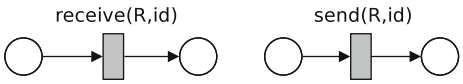

Fig. 10 The communication primitives: blocking receive and non-blocking send

whose execution time is bounded and significantly lower than action execution cycle. When this is not guaranteed, the current formulation should be extended with more complex forms of synchronization that use a non-instantaneous model of communication, but this aspect is beyond the scope of the present article.

The two communication actions are $receive(R_s, id)$ and $send(R_r, id)$, where R_r is the robot that receives the message id and R_s is the robot sending it; id is a unique identifier for the synchronization operators (which can be obtained by enumerating the operators). When R_s performs the $send(R_r, id)$ action, it instantaneously and reliably sends the message id to R_r, which stores it as the pair $\langle R_s, id \rangle$ in a buffer. Pairwise, a $receive(R_s, id)$ is executed by R_r only if there is a pair $\langle R_s, id \rangle$ in the buffer. If this is the case, the action fires removing the pair from the buffer.

Distributed hard synchronization. Figure 11 shows the decomposition of a centralized plan using *h_sync*, Fig. 11a, into two single robot plans, Fig. 11b, where the top plan belongs to R_1 and the bottom one to R_2. The decomposed plans look very similar to the centralized one, but for the *h_sync* being replaced by two communication 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 $send(r, id)$ and $receive(r, id)$ 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 $send(R_1, id)$ in a separate thread, and then stop on the blocking receive $receive(R_2, id)$. When R_2 arrives on the other side of the table, it will perform a non-blocking $send(R_2, id)$ in a separate thread. At this point, both robots will have received the id message, and will move on lifting the table together.

Distributed soft synchronization. Consider now the example of *s_sync* shown in Fig. 12a, and the two plans derived from it in Fig. 12b. In this case, the *s_sync* operator $s\_sync(R_1, R_2)$, is decomposed in only two primitives: a non-blocking send and a blocking receive. The reason for the asymmetric decomposition is that the *s_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 its task (i.e., opening the door). The send message is non-blocking

because there is no need to wait for executing $enterRoom$. Nevertheless, robot R_2 is blocked at $receive(R_1, id)$ on the main thread, because it has to be sure that $openDoor$ has ended, before performing $enterRoom$.

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 9 (*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.

Notice that the execution of each $R_i.\mathcal{P}$ is accomplished as described in Sect. 4 and the execution of each $R_i.\mathcal{P}$ 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 $PNPs\ \{R_i.\mathcal{P} = \langle R_i.P, R_i.T, R_i.F, R_i.W, M_0, G\rangle\}$ 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 $R_i.\mathcal{P}$.

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

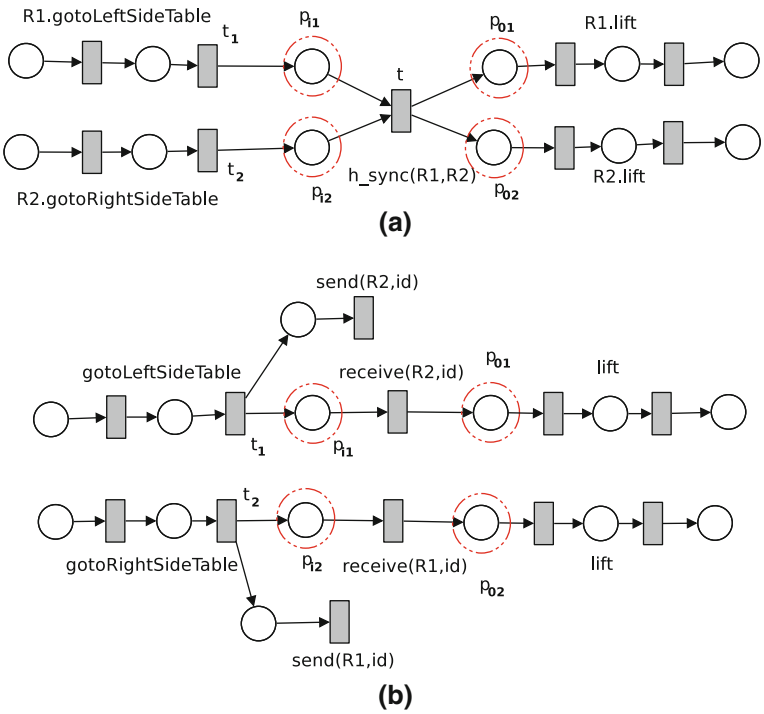

Fig. 11 **a** A multi-robot PNP using *h_sync*. **b** The single-robot PNPs obtained from the decomposition of the multi-robot one

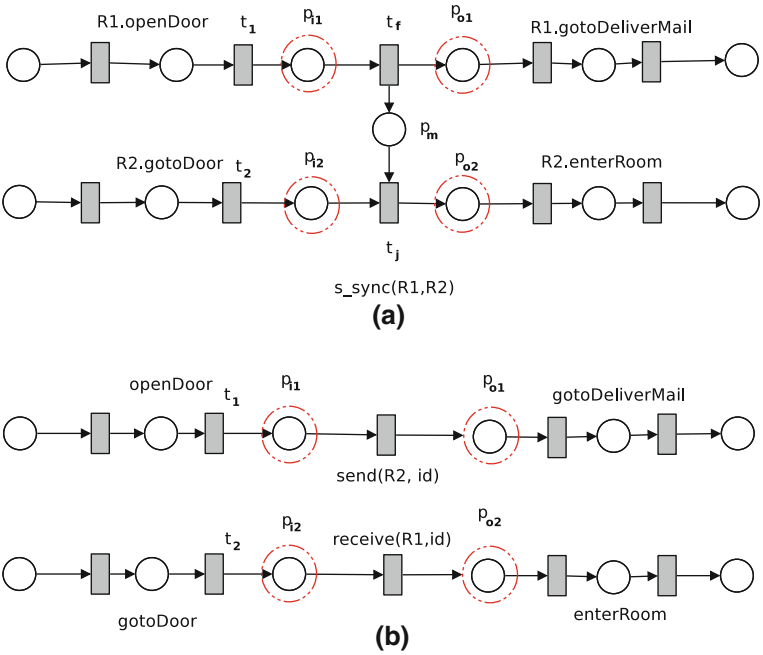

Fig. 12 **a** A multi-robot PNP using *s_sync*. **b** The single-robot PNPs obtained from the multi-robot one

of a net considering the marking of places, which actually control the behavior of the robot. This is obtained through the projection of the marking vector to a subspace, where places of synchronization operators (both centralized and distributed) are dropped. This allows us to verify if a centralized execution is equivalent to a distributed one, independently of the specific synchronization operator used.

Definition 10 (*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 projection of M_j, obtained by ignoring the places of multi-robot operators, such as hard and soft synchronizations for the case of a centralized execution, and of the communication primitives send and receive in the case of distributed execution.

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. During the execution of a multi-robot PNP, each robot executes the portions of the plan which are associated to it, except for *h_sync* and *s_sync* operators which require a central operator.

We show that synchronization operators have the same behavioral evolutions when centralized or distributed, by considering the behavior of the communication primitives $send(R_2, id)$ and $receive(R_1, id)$. All the communication primitives of a given decomposed operator will have the same id value, which identifies the operator itself.

Thus, if the distributed execution of the synchronization operators, when considering only their input and output places, has the same behavioral evolution of the centralized execution, then also the entire PNP has the same behavioral evolution if executed in a distributed or centralized way. The consequence of this is that, if a PNP is bounded, live or has a home

state, then also the distributed execution will be bounded, live and have the same homing state.

Theorem 2 *Assuming a reliable communication channel and instantaneous communication, DEA correctly executes any PNP $\mathcal{P}$ when using the multi-robot operators h_sync and s_sync, i.e. it produces the same behavioral evolution of the centralized execution of $\mathcal{P}$ with the same input.*

Proof Since synchronization operators do not involve queries to the knowledge base, we consider the behavioral evolution ignoring the *kb* term. In particular, we consider for both the hard and soft synchronization the marking (Figs. 8, 9):

$$\langle M(p_{i1}), M(p_{i2}), M(p_{o1}), M(p_{02})\rangle$$

Hard synchronization. Without loss of generality, consider the *h_sync* and its distributed counterpart depicted in Fig. 11a and b. Given that a PNP must be 1-bounded and that the underlying PN model prescribes at most one transition firing at every step, we have the following two possible scenarios for the centralized execution, based on the order with which transitions fire (Fig. 11a):

1. t_1, t_2: t_1 fires producing the marking $\langle 1, 0, 0, 0\rangle$, t is not enabled and thus no further firing can occur. Then, t_2 fires producing the marking $\langle 1, 1, 0, 0\rangle$. Now t is enabled an can fire producing the marking $\langle 0, 0, 1, 1\rangle$.
2. t_2, t_1: t_2 fires producing the marking $\langle 0, 1, 0, 0\rangle$, t is not enabled and thus no further firing can occur. Then, t_1 fires producing the marking $\langle 1, 1, 0, 0\rangle$. Now t is enabled an can fire producing the marking $\langle 0, 0, 1, 1\rangle$.

We show that the distributed execution has the same behavioral evolution (Fig. 11b):

1. t_1, t_2: t_1 fires producing the marking $\langle 1, 0, 0, 0\rangle$. At this point, the send action is instantaneously performed on a separate thread and no tokens are produced (notice that the token in the input place of the send action is not considered in the behavioral evolution because part of a send operator). The receive action can not fire because it is blocked. Then, t_2 fires producing the marking $\langle 1, 1, 0, 0\rangle$. The send action is instantaneously performed on a separate thread and no tokens are produced. Now, both receive actions have been unblocked, and can instantaneously terminate producing the marking $\langle 0, 0, 1, 1\rangle$.
2. t_2, t_1: The same applies here, because this case is the symmetric of the previous.

Soft synchronization. Without loss of generality, consider the *s_sync* and its distributed counterpart depicted in Fig. 12a and b. Given that a PNP must be 1-bounded and that the underlying PN model prescribes at most one transition firing at every step, we have the following two possible scenarios for the centralized execution, based on the order with which transitions fire (Fig. 12a):

1. t_1, t_2: t_1 fires producing the marking $\langle 1, 0, 0, 0\rangle$, t_f is enabled fires producing the marking $\langle 0, 0, 1, 0\rangle$ (notice that a token is also placed in p_m, which is not considered in the behavioral evolution because part of an operator). Then, t_2 fires producing the marking $\langle 0, 1, 1, 0\rangle$. Now t_j is enabled an can fire producing the marking $\langle 0, 0, 1, 1\rangle$.
2. t_2, t_1: t_2 fires producing the marking $\langle 0, 1, 0, 0\rangle$, t_j is not enabled and thus no further firing can occur. Then, t_1 fires producing the marking $\langle 1, 1, 0, 0\rangle$. Now t_f is enabled an can fire producing the marking $\langle 0, 1, 1, 0\rangle$. Finally, t_j fires producing the marking $\langle 0, 0, 1, 1\rangle$.

We show that the distributed execution has the same behavioral evolution (Fig. 12b):

1. t_1, t_2: t_1 fires producing the marking $\langle 1, 0, 0, 0\rangle$ and enables the send action which fires producing the marking $\langle 0, 0, 1, 0\rangle$. Then, t_2 fires producing the marking $\langle 0, 1, 1, 0\rangle$. Now, the receive action is enabled (and unblocked by the previous send action); it thus fires producing the marking $\langle 0, 0, 1, 1\rangle$.
2. t_2, t_1: t_2 fires producing the marking $\langle 0, 1, 0, 0\rangle$, but the receive action is blocked (because it did not receive the id message) and thus no further firing can occur. Then, t_1 fires producing the marking $\langle 1, 1, 0, 0\rangle$. Now, the send action is enabled and can fire producing the marking $\langle 0, 1, 1, 0\rangle$. Finally, the receive action, that has been unblocked by the send, fires producing the marking $\langle 0, 0, 1, 1\rangle$. □

6 Collaboration using PNPs

In this section, we describe the use of PNPs to model collaboration in a team of robots. Collaboration 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 h_sync operator. Nevertheless, h_sync and s_sync are not enough to model more general forms of explicit collaboration. To this end, we introduce a new synchronization mechanism, called joint committed action, that can be used to model explicit forms of collaboration such as Cohen and Levesque's Joint Intentions theory [6,51], also allowing for a distributed execution.

6.1 Task assignment

An example of multi-robot collaboration 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 [16], Market Based [14,60], Reactive Task Assignment [27,55], Iterative Task Assignment [42] or Sequential Task Assignment [5,13,23]). Among them, we consider the PNP implementation of a dynamic task assignment approach based on utility functions [27], 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. We implement the task assignment algorithm through an h_sync, where each send action communicates the id of the h_sync and its own computed utility for the task. The communicated utility allows for sharing knowledge on the task to be performed. Then, each robot 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 closer to the ball is typically required to execute the *pass* task, whereas the robot which is far from the ball should execute the *receive* task. A PNP that models task assignment in this scenario is shown in Fig. 13. 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.

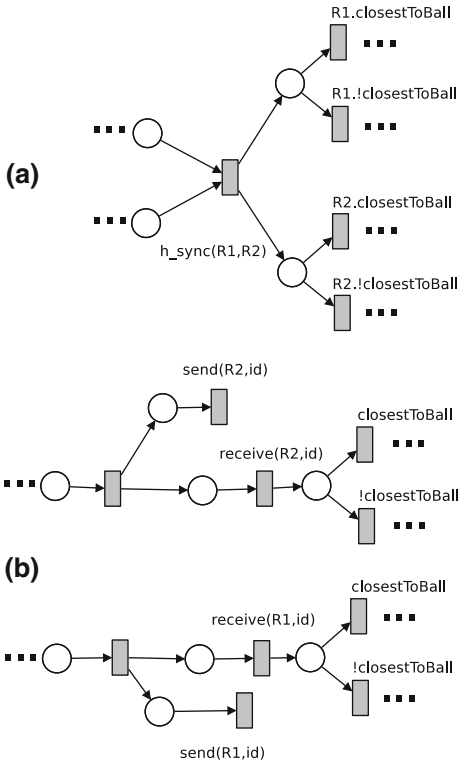

Fig. 13 PNP used for task assignment: **a** multi-robot plan and **b** single-robot plans

The task assignment is then consistently performed, in this case, through the evaluation of the condition $closestToBall$. In case $Robot1$ is the closest to the ball ($R1.closestToBall$ 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 considers 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 Joint intentions

Collaboration plays an important role in multi-robot systems, as teamwork can lead to consistent performance improvements. Cohen and Levesque's Joint Intentions (JI) theory [6] provides a detailed formal specification for the design of collaborative behaviors. Its prescriptive approach can be expressed using PNPs, which provide the required level of expressiveness, while maintaining the desired generality to allow for the design of a wide range of collaborative tasks. This section briefly summarizes the concepts behind the JI theory, and shows how it is possible to use it to design a PNP for explicit collaboration among multiple robots.

The Joint Intentions theory isolates a set of basic requirements that the collaborative behavior should fulfill. The theory is rooted in the concept of *commitment*: members that are committed to the execution of a collaborative behavior will continue their individual action execution until the commitment holds. Communication is used to achieve mutual belief about

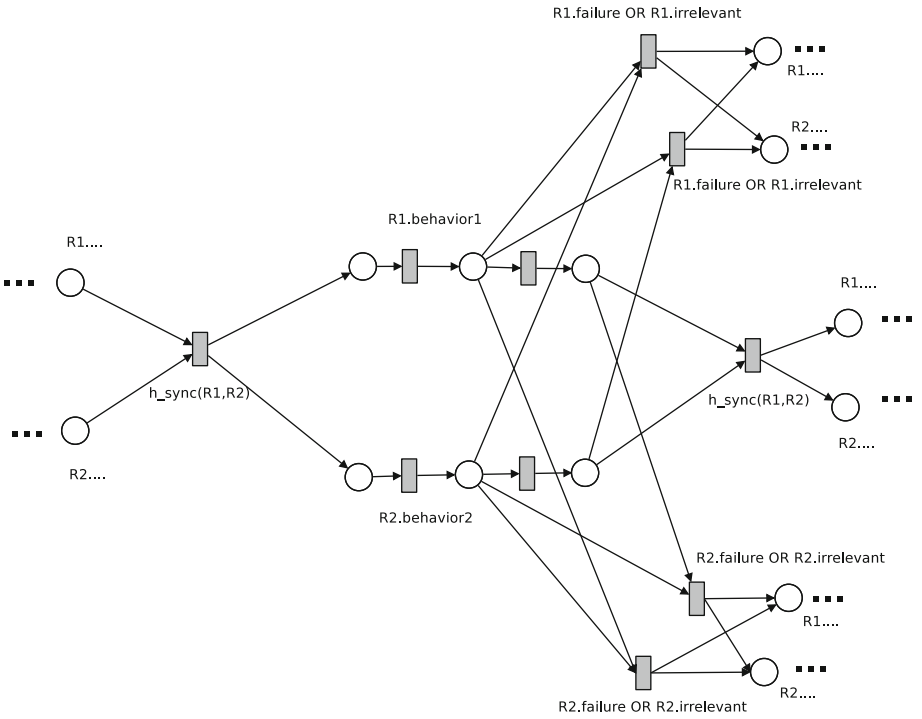

Fig. 14 The joint committed action. Transitions implicitly encode events associated to the change of the current state of the joint action. When distributed, mutual belief on the current state of the joint actions is established by sending ids that implicitly encode the state of the system. Individual robots commit to the execution of a collaborative behavior using an *h_sync* operator. In case all individual behaviors are successfully concluded the collaboration successfully terminates and the robots break their commitment thought a second *h_sync* operator. Individual robots are able to determine if the collaboration should be interrupted, and communicate to achieve mutual belief on the necessity to break their commitment

the necessity of interrupting the collaboration (i.e. breaking the members' commitment), in case any member of the team believes one of the following conditions is true:

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

The prescriptive approach of the Joint Intentions theory can be used to provide a systematic design of collaborative behaviors in a multi-robot team. The concepts behind the JI theory, in fact, can be embodied in the design of multi-robot PNPs for collaborative tasks. To this end, we define a *joint committed action* structure, which uses the h_sync operator to establish commitment and to assess the successful conclusion of the joint action. Moreover, we introduce a new set of transitions which act as multi-robot interrupts, to consistently interrupt the joint action when it becomes irrelevant or fails. Figure 14 shows the joint committed action: a multi-robot PNP operator for collaborative behavior, according to the specifications of the JI theory.

After a first hard synchronization (during which the commitment is established), the two robots perform the collaborative behavior, executing their individual actions ($behavior1$ and $behavior2$). In our implementations, the $behavior1$ and $behavior2$ are sub-plans rather than

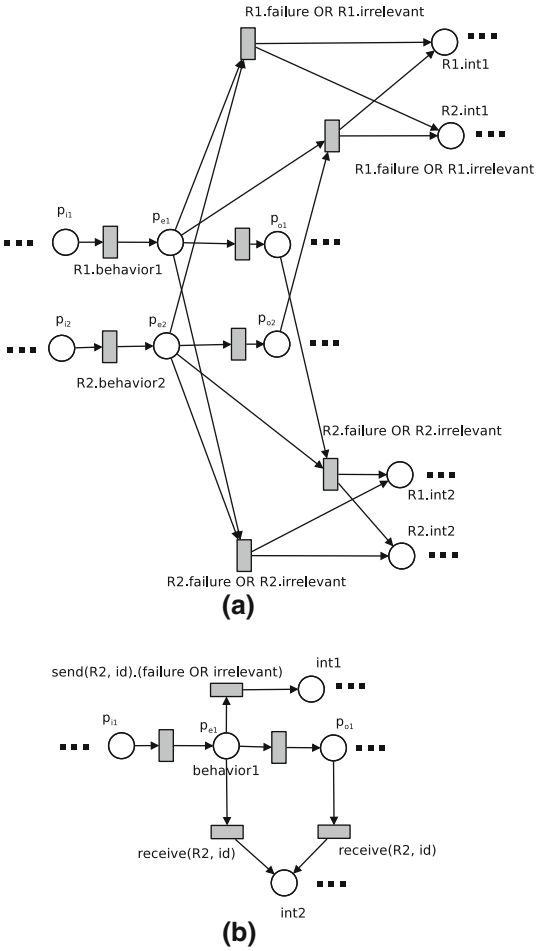

Fig. 15 **a** Portion of the Joint Committed Action ignoring h_sync. **b** The decomposed PNP for robot $R1$

single actions. This allows for higher modularity and for avoiding proliferation of multi-robot interrupts required to monitor all possible configurations of the execution state of parallel sequences of actions. Following the JI theory, the commitment is broken, if one of the above listed conditions holds. If one of the engaged robots senses that the action has become 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 the case of a successful termination of both $behavior1$ and $behavior2$, a hard sync is performed to successfully end the commitment.

The decomposition of a joint committed action amounts to decomposing the h_syncs and the transitions that act as multi-robot interrupt. The decomposition of the h_sync is performed as described in the previous section, hence we focus on the decomposition of the multi-robot interrupt (Fig. 15). Specifically, let us focus on the case when $R1$ recognizes that the joint committed action fails or becomes irrelevant (Fig. 16). In the centralized model, this situation is handled by two transitions which break the commitment and move the robots to some recovery plan. The two transitions are required to handle the two possible configurations in which the system may be: (1) both robots are executing their actions; (2)

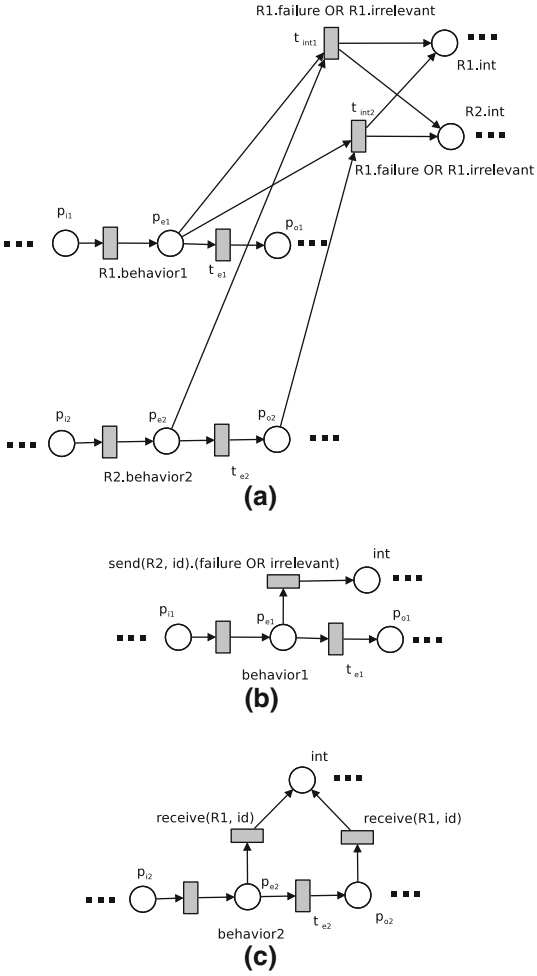

Fig. 16 Detail on the decomposition of a Joint Committed Action

$R1$ is currently executing its action, while $R2$ has finished his action. Notice that the two configurations depend on the state of $R2$. The decomposition can thus be obtained as follows. First, we add an instantaneous send action for $R1$ that acts as an interrupt for $behavior1$ and that is guarded by the interrupt conditions. The send action when performed interrupts $behavior1$ and starts the recovery procedure for $R1$. Second, we add two receive actions to $R2$, one for each possible configuration of $R2$, which are triggered by the message sent by $R1$. Both receive actions, which are mutually exclusive given the topology of the net, produce a token in the same place, which is the input place of some recovery procedure for $R2$. Notice that mutual knowledge is obtained through the communication primitives that implicitly encode the current state of the joint behavior in the ids. Specifically, ids relative to h_syncs encode the commitment to a joint action or the achievement of a goal, while ids of multi-robot interrupts encode failures or the irrelevance of a task.

The semantics of distributed execution described in Sect. 5 is extended to the joint committed action. As a consequence of Theorem 2 and of the correspondence of the behavioral evolution of the centralized and the distributed multi-robot interrupt, we can prove that DEA correctly executes any multi-robot PNP $\mathcal{P}$ which includes joint committed actions.

Theorem 3 *DEA correctly executes any PNP $\mathcal{P}$ when using joint committed actions, i.e. it produces the same behavioral evolution of the centralized execution of $\mathcal{P}$ with the same input.*

Proof The proof of this theorem is similar to the one for Theorem 2. The main difference is that the joint committed action requires the query of the local *kb* of a robot in order to identify the interrupt conditions. Given that the interrupt explicitly requires to specify on which Knowledge Base the interrupt condition must be verified, we can consider the behavioral evolution ignoring the *kb* term:

$$\langle M(p_{i1}), M(p_{i2}), M(p_{e1}), M(p_{e2}), M(p_{o1}), M(p_{o2}), M(R1.int), M(R2.int)\rangle.$$

Given that it has been shown that *h_sync* preserves behavioral evolutions (Theorem 2), in order to obtain the proof we discuss the case of the multi-robot interrupt. Specifically, given that the behavior of the interrupts is symmetrical, we consider the case of multi-robot interrupts detected on robot $R1$ (Fig. 16).

Given that a Joint committed always starts with a hard sync, we always begin with a token in the input place of both actions (i.e., $\langle 1, 1, 0, 0, 0, 0, 0, 0\rangle$). Moreover, given that the start of an action, if enabled, is automatically fired by the executor and that the executor's cycle time is orders of magnitude faster than any robot action, we assume that, independently of which actions fires first, the system will instantaneously transition to the marking (i.e., $\langle 0, 0, 1, 1, 0, 0, 0, 0\rangle$).

Consider now the multi-robot interrupt (Fig. 16a) and its distributed counterpart (Fig. 16b, c). For the centralized model, we have the following possible cases (Fig. 16a):

1. t_{e1} fires leading to the marking $\langle 0, 0, 0, 1, 1, 0, 0, 0\rangle$, then t_{e2} fires leading to the marking $\langle 0, 0, 0, 0, 1, 1, 0, 0\rangle$.
2. t_{e2} fires leading to the marking $\langle 0, 0, 1, 0, 0, 1, 0, 0\rangle$, then t_{e1} fires leading to the marking $\langle 0, 0, 0, 0, 1, 1, 0, 0\rangle$.
3. the condition (R1.failure OR R1.irrelevant) is true and t_{int1} fires leading to the marking $\langle 0, 0, 0, 0, 0, 0, 1, 1\rangle$.
4. t_{e2} fires leading to the marking $\langle 0, 0, 1, 0, 0, 1, 0, 0\rangle$, then the condition (R1.failure OR R1.irrelevant) is true and t_{int2} fires leading to the marking $\langle 0, 0, 0, 0, 0, 0, 1, 1\rangle$.

Let us consider the same situations, in the case of distributed execution (Fig. 16b, c):

1. t_{e1} fires leading to the marking $\langle 0, 0, 0, 1, 1, 0, 0, 0\rangle$, then t_{e2} fires leading to the marking $\langle 0, 0, 0, 0, 1, 1, 0, 0\rangle$.
2. t_{e2} fires leading to the marking $\langle 0, 0, 1, 0, 0, 1, 0, 0\rangle$, then t_{e1} fires leading to the marking $\langle 0, 0, 0, 0, 1, 1, 0, 0\rangle$.
3. the condition (R1.failure OR R1.irrelevant) becomes true, thus in the local net of R1 the guard of send(R2,id) is true and the send action fires sending the message id to $R2$. Simultaneously, the receive action connected to p_{e2} is enabled and unblocked, thus, it fires leading to the marking $\langle 0, 0, 0, 0, 0, 0, 1, 1\rangle$.
4. t_{e2} fires leading to the marking $\langle 0, 0, 1, 0, 0, 1, 0, 0\rangle$, then the condition (R1.failure OR R1.irrelevant) becomes true, thus in the local net of R1 the guard of $send(R2, id)$ is true and the send action fires sending the message id to $R2$. Simultaneously, the receive action connected to p_{o2} is enabled and unblocked, thus, it fires leading to the marking $\langle 0, 0, 0, 0, 0, 0, 1, 1\rangle$. □

Notice that if we drop the assumption of instantaneous communications and allow some bounded delay, the system may have some spurious evolutions. Specifically, the distributed

execution, before reaching the marking $\langle 0, 0, 0, 0, 0, 0, 1, 1\rangle$, may reach for some bounded amount of time the markings $\langle 0, 0, 0, 1, 0, 0, 0, 1\rangle$ or $\langle 0, 0, 1, 0, 0, 0, 0, 1, \rangle$. Despite this, the system will eventually reach the marking $\langle 0, 0, 0, 0, 0, 0, 1, 1\rangle$.

7 Evaluation

The proposed framework has been implemented and used to control different robotic systems in different domains. In particular, the implementation includes a plan executor for PNPs and 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.

In this section we first present three case studies showing the capabilities of the proposed framework and then we show some examples of PNP validation.

7.1 Case studies

Among several applications realized with our PNP framework, we describe experimental tests implemented on two different robotic platforms: a wheeled robot used for search and rescue missions and four legged AIBO robots used for robotic soccer. In particular, we describe three case studies: single-robot exploration and search in unknown environment, collaborative multi-robot foraging, and robotic soccer ball passing.

Our aim is to highlight the features of PNPs used for representing single-robot and multi-robot plans needed to accomplish them. The search and rescue activities described in [3] address the need of complex plan structures, like interrupts and concurrent execution of complex actions. The case study on multi-robot foraging [16] shows an example of complex coordination and action synchronization. Finally, the case study on robotic soccer presents many of the issues related both to coordination and collaboration. The videos showing the execution of these tasks with the robots are available at: pnp.dis.uniroma1.it.

7.1.1 Exploration and search in an unknown environment

Exploration and search in an unknown environment is an important task for many robotic applications, related to security and surveillance. The main difficulties arise from the impossibility of specifying all the possible situations that a robot will encounter during a mission. In this context, we put a special emphasis on the evaluation of different strategies for rescue robots [3] and we use PNPs to model the different strategies and perform a set of experiments with real and simulated robots to assess performance of such strategies.

Figure 17 shows the Pioneer 3 AT robot used in these experiments. It is a wheeled robot with on board sensors and computation. In particular, it uses a laser range finder for navigation and mapping, and a camera mounted on a pan-tilt unit for object detection and recognition. We also use a simulated scenario with similar characteristics.

While the description of the exploration strategies, the PNPs and the results of the experiments are detailed reported in [3], in this article we comment on the use of PNPs.

In the plans describing the exploration strategies we need to use all the single-agent PNP features described in the previous sections. Examples of ordinary actions are: *NavigateAndSearch*, *NavigateToCandidateVictim* and *AnalyzeVictim*; while sensing actions like *FireFound* are used to drive the plan according to some perception. The use of interrupts is very important, both to model action failures due to the uncertainty in perception and action execution and to take into account some other task more important to be performed.

Fig. 17 Rescue robots used for exploration and search tasks

For example, during the execution of the action *NavigateAndSearch*, that is the basic action used to move around the environment, if a victim is seen, an interrupt on a condition *newVictimFound* allows the robot to abort the search action and to start navigating towards the victim. Finally, concurrent actions are used in many situations to drive the robot wheels and the pan-tilt unit moving the camera independently.

In the above mentioned paper, we analyze four different scenarios and several strategies implemented with PNPs. PNPs embody all the features needed to model such a complex domain and the corresponding robot behaviors.

7.1.2 Coordination in robotic foraging

In the second 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 collaborative issues to an external algorithm. While the coordination algorithm, as well as a detailed description of the experiments, can be found in [16], in this paper, we describe the PNPs 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 (Fig. 18). 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.

Figure 19 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.

7.1.3 Collaboration in robotic soccer

The third case study shows a complete example of using PNPs, addressing both coordination and collaboration. The goal of the case study is to have two robots passing the ball to each

Fig. 18 AIBO robots during the foraging task

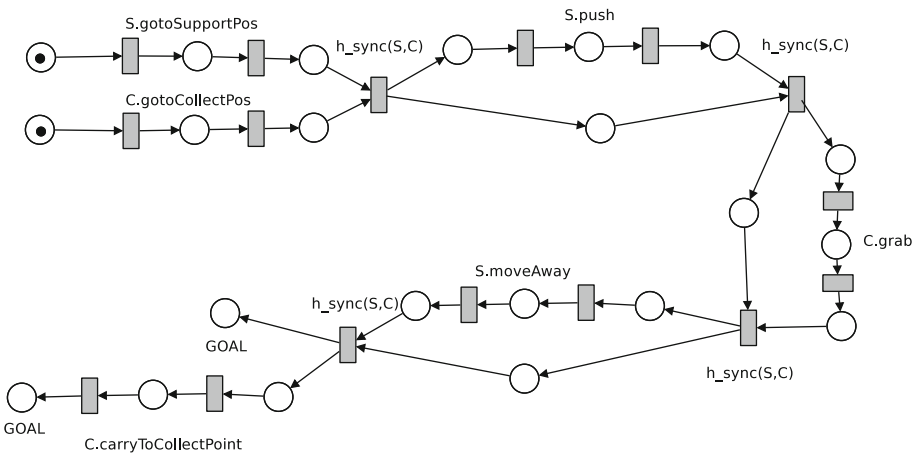

Fig. 19 The multi-robot PNP for foraging test

other (Fig. 20). We require the resulting behavior to be robust to both action failures and exogenous events. To this end, we need to model, through PNPs, action synchronization, dynamic task assignment and teamwork implemented through Joint Intentions theory. Thus, in contrast with the previous case study, here there is no external module for collaboration. Indeed, collaboration is accomplished by using the PNP structures described in the previous sections.

The multi-robot plan (Fig.21) 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 closer to the ball takes the role of the *Passer* and the other robot behaves as the *Receiver*. This is obtained by using the PNP task assignment mechanism illustrated by Fig. 13. Note that this assignment is dynamic and depends on the actual position of the ball.

Fig. 20 AIBO robots during the passing task

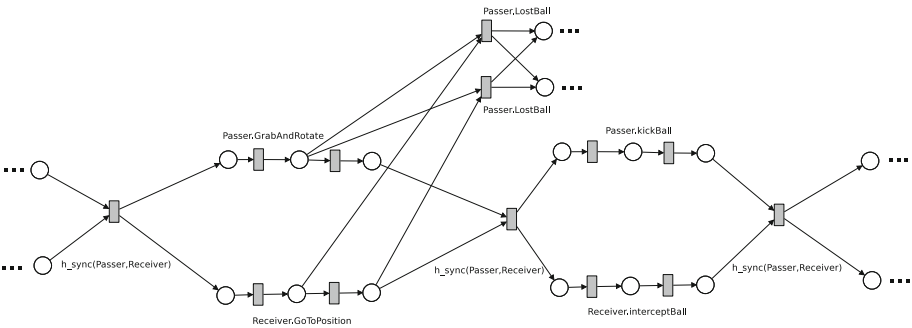

Fig. 21 Multi-robot PNP for the pass behavior

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_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 (Fig. 22) 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 JI theory, the robots break their commitment in case a failure occurs during this phase (in this particular task the collaborative behavior is never considered irrelevant, as the robots have the unique task of passing the ball). More specifically, the $Lost\,Ball$ 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*.

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

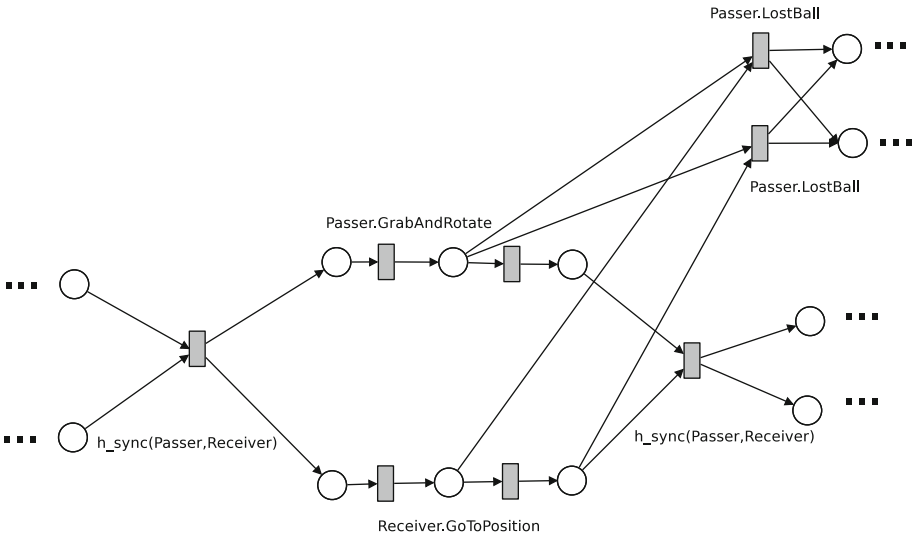

Fig. 22 Preparation phase of the pass behavior

sync operator) is performed to exchange information about the outcome of the behavior, and the commitment is broken.

7.2 Example of PNP validation

Figure 23 shows another possible experimental test, which combines the two previous ones: a collector robot C collects balls as they arrive, it passes the collected balls to a supporter robot S, that stores them somewhere. In this richer setting, it is possible to identify several design features that can be supported by the analysis of PNPs.

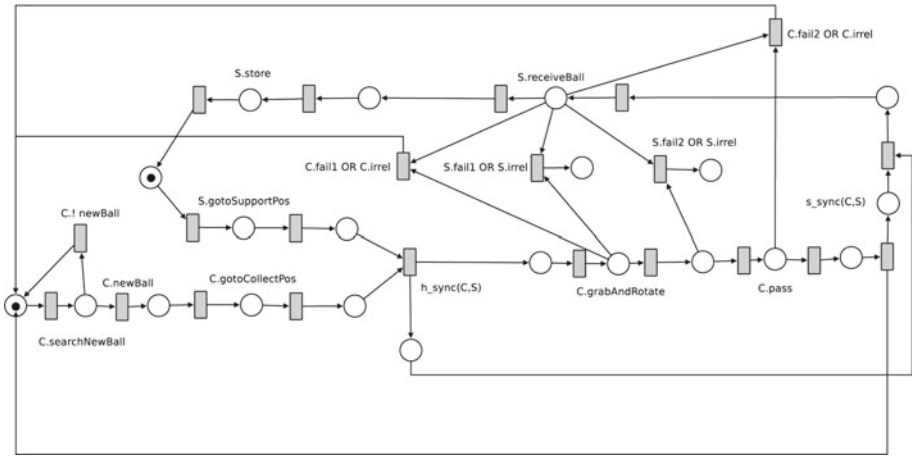

Fig. 23 A foraging task with pass behavior and commitment. The PNP in this example is not valid

– The 2-robot system can recover from commitment loss when the collector C fails, but not when the supporter S fails. This means that the net is not effective. The problem can be found through reachability analysis.
– 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 minimal, and the goal of continuously collecting balls is not achievable (will be undermined by a commitment break by S). The problem can be found through liveness analysis.
– If we replace the current *h_sync* with a *s_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_sync* place, meaning that the collector performs its task faster than the supporter, an undesirable feature. Thus the net is not safe. The problem can be found using coverability trees.

As this example shows, PN analysis tools can be used to study several features of PNPs. In general, these features may not only be limited to safeness, effectiveness and minimality as presented in Sect. 3.3. Moreover, we recall that once we verify that some properties hold for the centralized PNP, we also know that they will hold for the distributed version as a consequence of the fact that multi-robot operators preserve behavioral evolutions.

8 Conclusions

In this article, we have presented a new formalism, called PNPs, for high level programming of multi-robot systems. PNPs provide for a methodological and systematical approach for developing robotic behaviors through PNs, which allows for modeling many application-critical features such as: complex actions, concurrency and multi-robot coordination and collaboration. Moreover, PNPs implement a rich set of capabilities to support: (1) behavior design; (2) distributed execution; and (3) analysis and validation of the resulting behaviors. Our experience suggests that these features are very often required in the development of robotic systems, yet they can not be found in competing approaches.

PNPs are the result of the large body of experience in building multi-robot plans in a variety of domains, including robot soccer and disaster response robotics. The expressive power of PNPs provides suitable means to deal with most of the situations encountered, when designing autonomous robots and multi-robot systems. An implementation of the PNP executor, used in the case studies presented in this paper, is available as an open-source software. Additionally, PNPs can be supported by a wide range of PN tools that can ease the design and debug of PNPs through graphical interfaces and validate plans through automated analysis.

Although PNPs are the result of a decade of experience in designing multi-robot systems, there are still interesting developments in our research agenda. On the one hand, we need better ways to deal with the *symbol grounding* problem to establish a proper connection between low-level code and symbols in the plan representation framework: not only do we need to consider environment properties related to sensing actions and knowledge base predicates, but also we need better tools to define the action primitives and their counterpart in the low-level implementation. On the other hand, although PNPs are defined largely using notions from action theories, we aim at establishing a precise correspondence between PNPs and action theories as in the CONGOLOG language. In this way, the approach could also benefit from reasoning about actions as an additional tool to establish properties about states based upon the action representation. To this end, it is worth noticing that the combination of

constructs that is provided by PNPs is richer than the CONGOLOG language; for example, the inclusion of interrupts and commitments deserve further investigation.

Finally, another line of research could address the automatic generation or synthesis of PNPs. In this respect, it is possible to identify sublanguages of PNPs, with the goal of applying plan generation techniques to obtain PNPs from action representations, or to apply learning techniques (e.g., genetic programming or reinforcement learning) for refining or generating plans by experiments and user training.

References

1. Akharware, N. (2005). *Pipe2: Platform independent petri net editor*. M.Sc. thesis, Imperial College of Science, Technology and Medicine, University of London, London, UK.
2. Calisi, D., Censi, A., Iocchi, L., & Nardi, D. (2008, September). OpenRDK: a modular framework for robotic software development. In *Proceedings of international conference on intelligent robots and systems (IROS)*, pp. 1872–1877.
3. Calisi, D., Farinelli, A., Iocchi, L., & Nardi, D. (2007) Multi-objective exploration and search for autonomous rescue robots. *Journal of Field Robotics*, Special Issue on *Quantitative Performance Evaluation of Robotic and Intelligent Systems, 24*, 763–777.
4. Celaya, J. R., Desrochers, A. A., & Graves, R. J. (2007). Modeling and analysis of multi-agent systems using petri nets. In *IEEE international conference on systems, man and cybernetics (ISIC)*, pp. 1439–1444.
5. Chaimowicz, L., Campos, M. F. M., & Kumar, V. (2002, May). Dynamic role assignment for cooperative robots. In *Proceedings of the 2002 IEEE international conference on robotics and automation (ICRA02)*, pp. 292–298, Washington, DC
6. Cohen, P. R., & Levesque, H. J. (1991). Teamwork. *Special Issue on Cognitive Science and Artificial Intelligence, 25*, 486–512.
7. Coradeschi, S., & Saffiotti, A. (2003). An introduction to the anchoring problem. *Robotics and Autonomous Systems, 43*(2–3), 85–96.
8. Cost, R. S., Chen, Y., Finin, T., Labrou, Y. K., & Peng, Y. (2000). *Using colored petri nets for conversation modeling*, Vol. 1916 of *Lecture Notes in AI* (pp. 178–192). Berlin: Springer.
9. Costelha, H., & Lima, P. (2007). Modelling, analysis and execution of robotic tasks using petri nets. In *IEEE/RSJ international conference on Intelligent robots and systems (IROS)*, pp. 1449–1454, October 29–November 2, 2007.
10. De Giacomo, G., Iocchi, L., Nardi, D., & Rosati, R. (1997). Planning with sensing for a mobile robot. In *Proceedings of 4th European conference on planning (ECP'97)*.
11. De Giacomo, G., Lespérance, Y., & Levesque, H. J. (2000). Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence, 121*(1–2), 109–169.
12. de Silva, L., Sardina S., & Padgham, L. (2009). First principles planning in bdi systems. In *AAMAS '09: Proceedings of the 8th international conference on Autonomous agents and multiagent systems*, pp. 1105–1112. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2009.
13. Dias, M. B., & Stentz, A. T. (2001, August). *A market approach to multirobot coordination*. Technical Report CMU-RI-TR-01-26, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.
14. Dias, M. D., & Stentz, A. (2002, September) Opportunistic optimization for market-based multirobot control. In *2002 IEEE/RSJ international conference on Intelligent robots and systems (IROS'02)*, pp. 2714–2720.
15. Durfee, E. H. (1999). Distributed problem solving and planning. In G. Weiss (Ed.), *Multiagent systems: A modern approach to distributed artificial intelligence* (pp. 121–164). Cambridge: MIT Press.
16. Farinelli, A., Iocchi, L., Nardi, D., & Ziparo, V. A. (2006). 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. ISSN:0018-9219.
17. Ferber, J. (1999). *Multi-agent systems*. Boston: Addison-Wesley.
18. Fikes, R., & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2*, 189–208.
19. Firby, R. J. (1989). *Adaptive execution in complex dynamic worlds*. PhD thesis, Yale.

20. Gat, E. (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the tenth national conference on artificial intelligence*, pp. 809–815.
21. Gat, E. (1997, February). ESL: A language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the IEEE aerospace conference* (Vol. 1, pp. 319–324). Aspen, CO: Snowmass.
22. Georgeff, M. P., & Lansky, A. L. (1986). Procedural knowledge. In *Proceedings of the IEEE special issue on knowledge representation*, Vol. 74, pp. 1383–1398.
23. Gerkey, B., & Matarić, M. J. (2000, December). Principled communication for dynamic multi-robot task allocation. In *Proceedings of the international symposium on experimental robotics*, pp. 353–362, Waikiki, Hawaii.
24. Giordano, V., Ballal, P., Lewis, F., Turchiano, B., & Zhang, J. B. (2006). Supervisory control of mobile sensor networks: Math formulation, simulation, and implementation. *IEEE Transactions on Systems, Man and Cybernetics—Part B: Cybernetics, 36*(4), 554–562.
25. Gutnik, G., & Kaminka, G. A. (2006). Representing conversations for scalable overhearing. *Journal of Artificial Intelligence Research, 25*(1), 349–387.
26. Herrero-Perez, D., & Martinez-Barbera, H. (2010). Modeling distributed transportation systems composed of flexible automated guided vehicles in flexible manufacturing systems. *IEEE Transactions on Industrial Informatics, 6*(2), 166–180.
27. Iocchi, L., Nardi, D., Piaggio, M., & Sgorbissa, A. (2003). Distributed coordination in heterogeneous multi-robot systems. *Autonomous Robots, 15*(2), 155–168.
28. Kaminka, G. A., & Frenkel, I. (2005). Flexible teamwork in behavior-based robots. In *AAAI*, pp. 108–113.
29. King, J., Pretty, R. K., & Gosine, R. G. (2003). Coordinated execution of tasks in a multiagent environment. *IEEE Transactions on Systems, Man, and Cybernetics, Part A, 33*(5), 615–619.
30. Kobt, Y. T., Beauchemin, S. S., & Barron, J. L. (2007). Petri net-based cooperation in multi-agent systems. In *Proceedings of 4th Canadian conference on computer and robot vision*, 2007
31. Konolige, K. (1997). COLBERT: A language for reactive control in Saphira. *Lecture Notes in Computer Science, 1303*, 31–50.
32. Konolige, K., Myers, K. L., Ruspini, E. H., & Saffiotti, A. (1997). The Saphira architecture: A design for autonomy. *Journal of Experimental and Theoretical Artificial Intelligence, 9*(1), 215–235.
33. Kontes, G., & Lagoudakis, M. G. (2007). Coordinated team play in the four-legged robocup league. In *Proceedings of IEEE international conference on Tools with artificial intelligence (ICTAI)*, Vol. 1, pp. 109–116.
34. Kress-Gazit, H., Fainekos, G. E., & Pappa, G. J. (2009). Temporal logic-based reactive mission and motion planning. *IEEE Transactions on Robotics, 25*(6), 1370–1381.
35. Kuo, C.-H., & Lin, I.-H. (2006). Modeling and control of autonomous soccer robots using distributed agent oriented petri nets. In *IEEE international conference on Systems, man and cybernetics (SMC apos)*, Vol. 5, pp. 4090–4095.
36. Loetzsch, M., Risler, M., & Jungel, M. (2006). Xabsl—A pragmatic approach to behavior engineering. In *IEEE/RSJ international conference on Intelligent robots and systems, 2006*, pp. 5124–5129.
37. Lima, D., & Milutinovic, P. (2002). Petri net models of robotic tasks. In *IEEE international conference on Robotics and Automation (ICRA'02)*.
38. Maier, C., & Moldt, D. (2001). Object coloured petri nets—A formal technique for object oriented modelling. *Concurrent object-oriented programming and petri nets: Advances in petri nets*, pp. 406–427.
39. McCarthy, J., & Hayes, P. (1969). Some philisophical problems from the standpoint of artificial intelligence. *Machine Intelligence, 4*, 463–502.
40. Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE, 77*(4), 541–580.
41. Palamara, P. F., Ziparo, V. A., Iocchi, L., Nardi, D., Lima, P., & Costelha, H. (2008). A robotic soccer passing task using petri net plans (demo paper). In D. Parkes, J. P. Müller, L. Padgham, & S. Parsons (Eds.), *Proceedings of 7th international conference on Autonomous agents and multiagent systems (AAMAS 2008)* (pp. 1711–1712). Estoril, Portugal: IFAAMAS Press.
42. Parker, L. E. (1998). ALLIANCE: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation, 14*(2), 220–240.
43. Poutakidis, D., Padgham, L., & Winikoff, M. (1998). Debugging Multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of 1998 IEEE international conference on Systems, man and cybernetics*, San Diego, USA.

44. Rao, A. S., & Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, & E. Sandewall (Eds.), *Proceedings of the second international conference on Principles of knowledge representation and reasoning*. San Mateo: Morgan Kaufmann.
45. Reiter, R. (2001). *Knowledge in action: Logical foundations for describing and implementing dynamical systems*. Cambridge: MIT Press.
46. Russell, S. J., & Norvig, P. (2003). *Artificial intelligence: A modern approach* (2nd ed.). Singapore: Pearson Education.
47. Scherl, R., & Levesque, H. J. (1993). The frame problem and knowledge producing actions. In *Proceedings of the 11th national conference on Artificial intelligence (AAAI'93)*, pp. 689–695.
48. Sheng, W., & Yang, Q. (2005, July 24–28). Peer-to-peer multi-robot coordination algorithms: Petri net based analysis and design. In *Proceedings, 2005 IEEE/ASME international conference on Advanced intelligent mechatronics*, pp. 1407–1412.
49. Simmons, R., & Apfelbaum, D. (1998, October). A task description language for robot control. In *Proceedings of IEEE/RSJ international conference on Intelligent robots and systems (IROS)*, Vol. 3, pp. 1931–1937. Victoria, BC, Canada.
50. Sudeikat, J., Braubach, L., Pokahr, A., & Lamersdorf, W. (2006). Validation of bdi agents. In R. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni (Eds.), *The 4th international workshop on Programming multiagent systems (PROMAS-2006)* (pp. 185–200). Berlin: Springer.
51. Tambe, M. (1997). Towards flexible teamwork. *Journal of Artificial Intelligence Research, 7*, 83–124.
52. Thrun, S., Burgard, W., & Fox, D. (2005). *Probabilistic Robotics (Intelligent robotics and autonomous agents)*. Cambridge: The MIT Press.
53. Vishwanadham, N., & Narahari, Y. (1992). *Performance modelling of automated manufacturing systems*. New Delhi: Prentice Hall.
54. Wang, F. Y., Kyriakopoulos, K. J., Tsolkas, A., & Saridis, G. N. (1993). A petri-net coordination model for an intelligent mobile robot. *IEEE Transactions on Robotics and Automation, 9*(3), 257–271.
55. Werger, B. B., & Mataric, M. J. (2000). Broadcast of local eligibility for multi-target observation. In *DARS00*, pp. 347–356.
56. Xu, D., Volz, R., Ioerger, T., & Yen, J. (2002). Modeling and verifying multi-agent behaviors using predicate/transition nets. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering* (pp. 193–200), New York, NY: ACM.
57. Zimmermann, A., & Freiheit, J. (1998). TimeNETMS-an integrated modeling and performance evaluation tool for manufacturing systems. In *Proceedings of 1998 IEEE international conference on Systems, man and cybernetics*. San Diego, USA.
58. Ziparo, V. A., & Iocchi, L. (2006). Petri net plans. In *Proceedings of fourth international workshop on modeling of objects, components, and agents (MOCA)*, pp. 267–290, Turku, Finland. Bericht 272, FBI-HH-B-272/06.
59. Ziparo, V. A., Iocchi, L., Nardi, D., Palamara, P. F., & Costelha, H. (2008). Petri net plans: a formal model for representation and execution of multi-robot plans. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems* (pp. 79–86). Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.
60. Zlot, R., Stenz, A., Dias, M. B., & Thayer, S. (2002). Multi robot exploration controlled by a market economy. In *IEEE international conference on robotics and automation (ICRA'02)*, pp. 3016–3023.