Auton Robot (2012) 33:337–360
DOI 10.1007/s10514-012-9288-x

Robot task plan representation by Petri nets: modelling, identification, analysis and execution

Hugo Costelha · Pedro Lima

Received: 20 May 2011 / Accepted: 10 March 2012 / Published online: 29 March 2012

Abstract In this paper we introduce a framework to represent robot task plans based on Petri nets. Our approach enables modelling a robot task, analysing its qualitative and quantitative properties and using the Petri net representation for actual plan execution. The overall model is obtained from the composition of simple models, leading to a modular approach. Analysis is applied to a closed loop between the robot controller and the environment Petri net models. We focus here on the quantitative properties, captured by stochastic Petri net models. Furthermore, we introduce a method to identify the environment and action layer parameters of the stochastic Petri net models from real data, improving the significance of the model. The framework building blocks and a single-robot task model are detailed. Results of a case study with simulated soccer robots show the ability of the framework to provide a systematic modelling tool, and of determining, through well-known analysis methods for stochastic Petri nets, relevant properties of the task plan applied to a particular environment.

Keywords Analysis · Execution · Identification · Modelling · Petri nets · Robot tasks

H. Costelha · P. Lima (✉)
Institute for Systems and Robotics, Instituto Superior Técnico, Lisbon, Portugal
e-mail: pal@isr.ist.utl.pt

H. Costelha
e-mail: hcostelha@isr.ist.utl.pt

H. Costelha
Department of Electrical Engineering, School of Technology and Management, Polytechnic Institute of Leiria, Leiria, Portugal

1 Introduction

Robots are becoming part of our daily lives, and their tasks are becoming more complex. Qualitative and quantitative formal methods to design and analyse these tasks are needed to ensure that relevant properties are met (Akin et al. 2008). Robot task models not based on formal approaches tend to be tailored to the task at hand, usually leading to task plans with few actions. Applying formal methods to model robot tasks provides a systematic approach to modelling, analysis and design, scaling up to realistic applications, and enabling analysis of formal properties, as well as design from specifications.

An interesting class of formal approaches is that of Discrete Event Systems, with two main formalisms being used in the Robotics community: Finite State Automata (FSA) (Cassandras and Lafortune 2008) and Petri nets (Petri 1966; Murata 1989). Most FSA-based models are used for robot tasks design and execution, although they can also be used to perform quantitative (Košecká et al. 1997) and qualitative analysis (Espiau et al. 1995), or for plan verification and code generation (Basu et al. 2008). Petri nets have advantages with respect to FSA: Petri-net marked languages are a superset of regular languages (marked by FSA), leading to a larger modelling power. Furthermore, one can model infinite state spaces with finite Petri nets and, in general, for the same state space dimension, the size of a Petri net is smaller than that of an FSA. Moreover, although composition of Petri nets usually leads to an exponential growth in the state space (as for FSA), structurally the growth is linear in the size of the composed graphs given that the state is distributed. State distribution makes the design process simpler for the task designer, due to its modularity and compositional nature—complex models can be obtained from the composition of simple models designed for the system basic components—and helps managing the display of the tasks both for monitoring and design purposes.

Petri nets have been widely used to model and control Flexible Manufacturing Systems (FMS) (Viswanadham and Narahari 1992; Zhou and Venkatesh 1999). FMS are usually well-structured systems, even if they can be large and complex. An FMS is composed of subsystems with clearly defined functions, inputs and outputs, most events are controllable, and uncontrollable events are often associated with machine failures only (Castelnuovo et al. 2007; Qin and Xu 2009; Viswanadham and Narahari 1992; Zhou and Venkatesh 1999). In a recent paper Herrero-Perez and Martinez-Barbera (2008) used Petri nets to model and control an FMS with Autonomous Guided Vehicles (AGV). A collision-free motion of the AGV is ensured by using a topological map and preventing more that one AGV at each node. Although untimed Petri nets are used, the authors assume that each node travelled in the topological map has an associated time, and minimise the number of nodes travelled to compute the task plan which minimises task time. In this system AGVs do not interact with each other and failures are not modelled. In contrast, non-industrial robot tasks (e.g., search and rescue, home assistance, soccer and other games) require plans whose actions can have several uncertain effects, due to the occurrence of many possible uncontrollable events, very dynamic (and even adversarial) environments, and unpredictable causes. Therefore, though the analysis methods may be similar, the models required are significantly different. Marked Ordinary Petri Nets (MOPNs) were first used to model and control a robot task plan execution in Wang et al. (1991), ensuring qualitative properties by construction. Performance properties were determined through simulations. More recently, Kim and Chung (2007) used Generalised Stochastic Petri Nets (GSPNs) to model and analyse (both qualitatively and quantitatively) a robot task for a tour-guide robot. However, the authors approach is very application-oriented and has not provided a structured framework for modelling and analysing generic robot tasks. Furthermore, there is no clear distinction between the selection mechanisms and uncontrollable events induced by the environment, leading to a less modular design. Multi-agent and multi-robot system models can also benefit from the modular nature of Petri nets. In King et al. (2003) MOPNs are used to model and synthesise deadlock-free plans for a multi-agent environment. However, the authors do not model failures, and uncontrolled events are modelled using a simplistic approach by considering that the probability of resources availability change does not depend on the current state. Furthermore time is not taken into account, as all transitions are considered to be instantaneous. In Ziparo and Iocchi (2006) Petri nets are used to design (multi-)robot tasks plans, providing qualitative (logical) but not quantitative (performance) analysis.

This paper introduces an integrated framework based on Petri nets for modelling, identification, analysis and execution of robot tasks, intended to overcome the shortcomings of previous approaches:

– the framework models single and multi-robot tasks;
– MOPN and GSPN views of the model are used to retrieve logical/qualitative and (probabilistic) performance/quantitative properties of the robot task plans, respectively;
– controllable (e.g., decision to start an action) and uncontrollable (e.g., failure to track a moving object, reaching a given location, actions of other robots) events are modelled;
– the GSPN view models action effects uncertainty both as plain transition probabilities (as in Markov Decision Processes) and as stochastic timed transitions, where transition probabilities are indirectly modelled by the stochastic time elapsed between the start of the action and its end, due to some uncontrollable event; in the end, both models boil down, under some light requirements, to an equivalent Markov Decision Process, that can be solved using existing techniques, from dynamic programming to reinforcement learning;
– the complexity of non-industrial robot tasks is tackled by creating a Petri net model of the environment, capturing the complexity of the environment dynamics: this model is then composed with the (multi-)robot controller model to obtain a single closed-loop Petri net representing the whole task model, i.e., the model of the (multi-)robot system situated in its environment; furthermore, an identification algorithm is introduced to estimate the parameters of the environment model from real data.

The robot task model is split in several layers, all Petri net-based, ranging from the organisation of the robot actions to the action and environment models. This structured approach organises the model construction based on formal definitions, starting from simple models of the (multi-)robot controller and environment components which are then automatically composed by well-defined algorithms. The resulting Petri net is used to compute qualitative and quantitative properties of the task using the adequate view of the model and available Petri net analysis algorithms.

This paper describes the single-robot part of the framework, but not the multi-robot one, due to space constraints. Nevertheless, extending the framework to the multi-robot case consists mostly on the introduction of communication models and the use of these to synchronise robot behaviours, similarly to the ones introduced in Costelha and Lima (2008), plus selection mechanisms, as further discussed in Sect. 9.

This paper is organised as follows: Sect. 2 introduces the Petri net models used in this work (for completeness); Sect. 3 describes the model building blocks; Sect. 4 explains the layered architecture and how task models can be designed based on these layers; in Sect. 5 we describe how to perform analysis of robot tasks based on our model; Sect. 6 details how environment models can be identified from real data; Sect. 7 explains how plans are executed within this framework; finally, Sect. 8 provides results obtained using a realistic physics simulator of robot soccer, and Sect. 9 draws the conclusions and future work.

2 Petri nets

Petri nets (Petri 1966) are a widely used formalism for modelling Discrete Event Systems. They allow modelling important aspects such as synchronisation, resources availability, concurrency, parallelism and decision making, providing at the same time a high degree of modularity, making them suitable to model robot tasks. Modularity in Petri nets is achieved since each resource can be modelled separately and then composed with others. Although composition operators exist for FSA, Petri nets can model subsystems with input and output places, which can be connected as in a circuit.

2.1 Marked ordinary Petri nets

The simplest models we use are MOPNs:

Definition 1 A MOPN is a five-tuple $PN = \langle P, T, I, O, \mathcal{M}_0\rangle$, where (Murata 1989; Cassandras and Lafortune 2008):

- $P = \{p_1, p_2, \dots, p_n\}$ is a finite, not empty, set of places;
- $T = \{t_1, t_2, \dots, t_m\}$ is a finite set of transitions;
- $I = P \times T$ represents the arc connections from places to transitions, such that $i_{rs} = 1$ if, and only if, there is an arc from p_r to t_s, and $i_{rs} = 0$ otherwise;
- $O = T \times P$ represent the arc connections from transition to places, such that $o_{rs} = 1$ if, and only if, there is an arc from t_r to p_s, and $o_{rs} = 0$ otherwise;
- $\mathcal{M}_j = [m_1(j), \dots, m_n(j)]$ is the state of the net, and represents the marking of the net at time j, where $m_n(j) = q$ means there are q tokens in place p_n at time instant j. $\mathcal{M}_0$ is the initial marking of the net.

The state of the net is given by the marking of the net, which in turn is given by the number of tokens in the places. In this class of Petri nets, all the transitions are *immediate* (have zero firing time), i.e., once they are enabled, they are fired and the new marking is instantly reached. A transition is enabled when all its input places have at least one token.

2.2 Generalised stochastic Petri nets

MOPNs are suited for qualitative analysis, but not for quantitative performance analysis. For this purpose, one uses generalised stochastic Petri nets (GSPNs). GSPNs include 2 transition types:

- stochastic (with probabilistically distributed time) transitions which, once enabled, fire only when a given time d has elapsed. In this work, we assume exponentially distributed time, and univocally define the exponential distribution associated to each transition by its rate μ;
- immediate transitions which, once enabled, fire in zero time. However, if more than one transition is enabled in a given marking, a probability mass function is associated to the set of conflicting transitions, such that, when such a marking is reached, the transition to be fired is picked randomly according to that probabilistic distribution.

These 2 transition types enable distinct models of uncertainty. Stochastic transitions allow modelling uncertainty according to the different stochastic time-to-fire distributions associated to conflicting transitions. If no information is available about time, one uses immediate transitions and directly associate probabilities to conflicting transitions.

Definition 2 A GSPN is an eight-tuple $PN = \langle P, T, I, O, \mathcal{M}_0, R, W\rangle$, where (Viswanadham and Narahari 1992; Bause and Kritzinger 2002):

- $P, T, I, O, \mathcal{M}_0$ are as defined in Definition 1;
- T is partitioned in two sets: T_I of immediate transitions and T_E of exponential transitions;
- R is a function from the set of transitions T_E to the set of real numbers, $R(t_{E_j}) = \mu_j$, where μ_j is called the firing rate of t_{E_j};
- W is a function from the immediate transitions set T_I to a set of real numbers, $W(t_{I_j}) = w_j$, where w_j is the weight associated with immediate transition t_{I_j};
- For any given marking, the probability of firing an enabled transition $t_I \in T_I$ is equal to $w_i/\mathcal{W}$, where $\mathcal{W}$ is the sum of the weights of all enabled transitions for the marking; the probability of firing an enabled transition $t_E \in T_E$ is equal to $\mu_i/\mathcal{M}$, where $\mathcal{M}$ is the sum of the firing rates of all enabled transitions for the marking.

Consider the GSPN model depicted in Fig. 1. In this example, t_{E_1} is an exponential timed transition (drawn as an unfilled rectangle), while t_{I_1}, t_{I_2} and t_{I_3} are immediate transitions with associated weights. Initially t_{E_1} is enabled, since p_1 has tokens, and will fire after an exponentially distributed time with rate μ_1 has elapsed. The token flows from p_1 to p_2 and, since t_{I_1} is an immediate transition, it will immediately flow from p_2 to p_3, reaching marking $\mathcal{M}_3 = [0, 0, 1]$. In this marking t_{I_2} and t_{I_3} form a set of

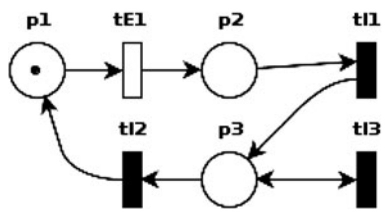


Fig. 1 Generalised stochastic Petri net

conflicting transitions, whereas only one will fire, according to the following probabilities:

$$P_f(t_{I_2}) = \frac{w_2}{w_2 + w_3}, \qquad P_f(t_{I_3}) = \frac{w_3}{w_2 + w_3} \tag{1}$$

If t_{I_3} is fired, the marking remains the same, if t_{I_2} is fired, the net returns to the initial marking.

The GSPN marking is a semi-Markov process with a discrete state space given by the reachability graph of the net for an initial marking (Murata 1989; Viswanadham and Narahari 1992). A Continuous Time Markov Chain (CTMC) and/or the corresponding Embedded Markov Chain (EMC) can be obtained from the marking process, and both the transition rate matrix (CTMC) and the transition probability matrix (EMC) can be computed by using the firing rates of the exponential timed transitions and the probabilities associated with the random switches. With these one can perform transient and stationary analysis of the chain, thus obtaining performance properties for the corresponding Petri net model.

3 Framework building blocks

In our framework, we endow the Petri net models with some additional building blocks and make use of the place labels to distinguish between different types of places, such as: *predicate places*, *action places*, *task places* and *regular* (or *memory*) *places*. These different types of places do not introduce any change regarding the Petri nets definitions given in the previous section, but are crucial in the analysis process explained later.

3.1 Predicate places

Predicate places are used to represent knowledge through logic predicates, having always one or zero tokens. Although Predicate Petri nets exist in the literature (Röck and Kresman 2006), the tools available to work with this type of Petri nets are scarce. As such, we use regular places to represent predicates, as explained next.

Definition 3 A *predicate place* p_n is a place associated with the predicate $\mathcal{P}$, described by $p_n \models \mathcal{P}$, such that:

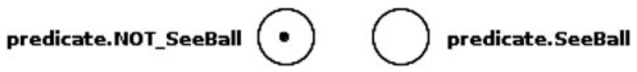


Fig. 2 Representation of predicate by a set of places

- $\forall_j, \mathcal{P}(j) = \mathit{true} \Leftrightarrow m_n(j) = 1$,
- $\forall_j, \mathcal{P}(j) = \mathit{false} \Leftrightarrow m_n(j) = 0$,
- $\forall_j$, if p_i is a place associated with predicate $\neg\mathcal{P}$ then $m_i(j) + m_n(j) = 1$,

where $\mathcal{P}(j)$ is the predicate $\mathcal{P}$ at time step j.

As an example, a Petri net model representing the predicate `SeeBall` is depicted in Fig. 2. Note the usage of the "*predicate.*" (or, alternatively, "*p.*") prefix to denote that the place is a predicate place, and the "*NOT_*" prefix to denote the negated predicate.

When executing tasks in the real and simulated robots, the marking of the predicate places is updated by monitoring the environment conditions, using the robot sensors, while when performing theoretical analysis it will be updated using environment Petri net models. In this paper, predicate places will always represent information obtained from sensor data although, as stated in Sect. 9, these can be used for other purposes.

3.2 Action and action places

An action is the elementary block on the execution of a task by a robot. The execution of an action changes the world, depending on the current state. Action places are elementary blocks in the Petri net models of a task plan. For execution purposes they are an atomic element, and will not be decomposable. For analysis purposes they will act as macro places, thus corresponding to a Petri net model of the action. Macro places (Bernardinello and Cindio 1992), albeit not always using the same definition, are used to create layered Petri nets, leading to a higher degree of modularity. The formal definition of action places will be given in Sect. 4.3.

3.3 Task plans and task places

A task plan is a network of actions and other task plans, where each action/task plan is selected according to the state of the world. In this framework task plans will be modelled using Petri nets, with task places being also macro places. This approach will allow to draw entire Petri net models from lower layers as single places in higher layers, providing for modular and reusable models. The formal definition of task places will be given in Sect. 4.3.

3.4 Memory places

All places which are not predicate, task or action places, are regular (or memory) places, i.e., normal Petri net places with

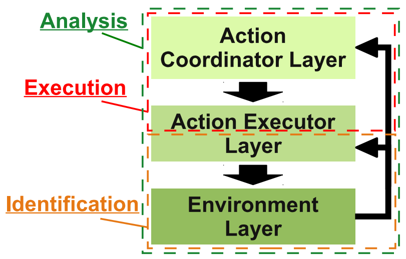


Fig. 3 Model architecture, pointing out the 3 layers and where each of them intervenes in the analysis, execution and identification.

no specific properties or associations. These are denoted as memory places given that they are mainly used to store some information on past conditions.

4 Modelling single-robot tasks using Petri nets

To achieve the desired modelling goals (modularity, design, analysis and execution), a 3-layered Petri net model of a robot task is introduced, as depicted in Fig. 3. Each layer, composed of a set of Petri net models representing different resolution levels, is described as follows:

Environment Layer Petri net models in this layer represent the environment discrete-state event-driven dynamics, resulting from the robot and other agent (robots, humans) actions or from physics (such as the motion of a free rolling ball);

Action Executor Layer Petri net models in this layer represent actions, i.e., the changes in the environment caused by these actions, and the conditions under which these changes can occur;

Action Coordinator Layer Petri net models in this layer represent task plans, which consist of compositions of actions.

As can be seen in Fig. 3, all models are used for analysis, but only the 2 top layers and, partially, the Action Executor layer models are used for execution. The Environment layer and part of the Action Executor layer, namely regarding the actions effects on the environment, are used for analysis. These represent models of real or simulated robots that can be parametrised beforehand through an identification algorithm based on real or simulated data, as explained in Sect. 6. These models are replaced by either a realistic simulation of the robots and environments, or the actual robots and sensor readings, when executing the Petri nets of the upper levels.

The use of discrete event system models for the environment brings loss of accuracy and some properties that can not be checked, due to the continuous state space nature of the environment. Hybrid systems would be more appropriate to increase accuracy and model-checking capabilities (Kress-Gazit et al. 2009). However, the complexity of their

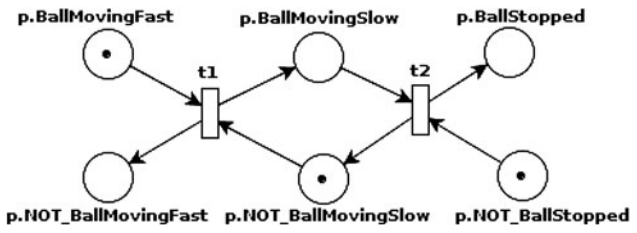


Fig. 4 Petri net model of a moving ball

analysis grows substantially with the number of actions and predicates, and many aspects of robot tasks, aside of robot motion, intrinsically happen in discrete state space and are event-driven.

4.1 Environment layer

The Environment layer includes models of the environment dynamics, caused by the controlled robot, other agents, or simply by the laws of physics.

Definition 4 An *Environment Petri net model* is a GSPN, where:

1. P contains only predicate or memory places;
2. If there is an arc from place p_n, associated to predicate $\mathcal{P}$, to transition t_j, then there is an arc from t_j to place p_m, associated to predicate $\neg\mathcal{P}$, or an arc back to p_n.

Definition 4 is very generic, only limiting the structure of the Petri net so that it conforms with Definition 3, regarding predicate places. It considers that transitions are used either to test a predicate value (place as input and output of the transition), or to change that predicate value from $0/1$ to $1/0$, keeping the consistency of predicate places as defined in Definition 3. To better understand how the Environment models are designed, consider a free rolling ball. In this case, due to friction on the floor, it is expected that the ball will stop after some time, with that time being stochastic (it depends on unknown conditions such as the ball fill pressure, floor, etc.). Figure 4 shows a possible GSPN model of this process under our framework. To create this model, we must first discretise the state space, such that we can describe the process through the use of logic predicates. In this example, we consider that the ball could be moving fast (`p.BallMovingFast`), slowly (`p.BallMovingSlow`) or be stopped (`p.BallStopped`), and that the ball will, with time, evolve from its fastest speed to the stopped state, by firing transition t_1 and later transition t_2. The stochastic transitions in the GSPN model allow modelling the stochastic nature of the events. Each transition will fire after being enabled by a stochastic amount of time, with transition t_1 leading from state {`p.BallMovingFast`, `p.NOT_BallMovingSlow`, `p.NOT_BallStopped`}

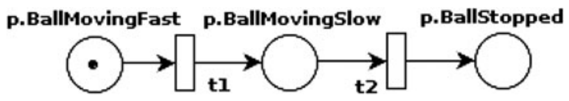


Fig. 5 Simplified Petri net model version of a moving ball

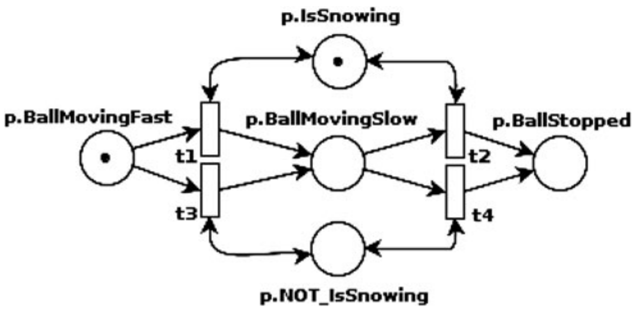


Fig. 6 Petri net model of a moving ball considering the weather conditions

to state {`p.NOT_BallMovingFast`, `p.BallMovingSlow`, `p.NOT_BallStopped`}, and transition t_2 from that state to state {`p.NOT_BallMovingFast`, `p.NOT_BallMovingSlow`, `p.BallStopped`}.

Since predicate places must follow Definition 3, there is no need to always draw both positive and negative forms of the predicate. In practise, the models can be drawn in a simpler form, such that the missing nodes are added during the analysis part (as explained in Sect. 5.2.1). For instance, the model depicted in Fig. 4 can be drawn as shown in Fig. 5.

If, for instance, one also wanted to model the fact that some other agent could increase the ball speed, we could add transitions in the opposite direction, albeit with different associated rates, considering the probability of that occurrence. Furthermore, it is also possible to include several transitions with different rates, associated with the same state change, as in the example depicted in Fig. 6. In this example, the rate at which the ball slows down depends on the weather conditions.

4.2 Action executor layer

Each action Petri net model is a GSPN which represents how the action impacts the environment and under which conditions. In logical terms, these models use two sets of conditions:

running-conditions Conditions that need to be true for the action to be able to produce any effect;
desired-effects Conditions the action aims to make true.

Each action model consists of several places and of a set of transitions representing the environment changes, which can be associated with the success or failure of the action, following the rules described in Definition 5 below. The general model of an action is depicted in Fig. 7.

Definition 5 An *action Petri net model* is a GSPN, where:

1. $P = P_E \cup P_R$ contains only predicate places, where

   P_E is the *effects* place set;
   P_R is the *running-conditions* place set;

2. $P_E = P_{E_S} \cup P_{E_F}$, where P_{E_S} and P_{E_F} are designated respectively *success places set* and *failure places set*.
3. $P_{E_S} = P_{E_{S_I}} \cup P_{E_{S_D}}$, where $P_{E_{S_I}}$ and $P_{E_{S_D}}$ are designated as *intermediate effects place set* and *desired-effects place set*, respectively;
4. $T = T_S \cup T_F$ with $T_S \cap T_F = \emptyset$, where:

   T_S is the set of transitions associated with successful impact of the action;
   T_F is the set of transitions associated with failure impact of the action;

5. If there is an arc from place p_n, associated to predicate $\mathcal{P}$, to transition t_j, then there is an arc from t_j to place p_m, associated to predicate $\neg\mathcal{P}$, or an arc back to p_n;
6. All transitions have one input arc from each *running-condition*.

A note on notation:

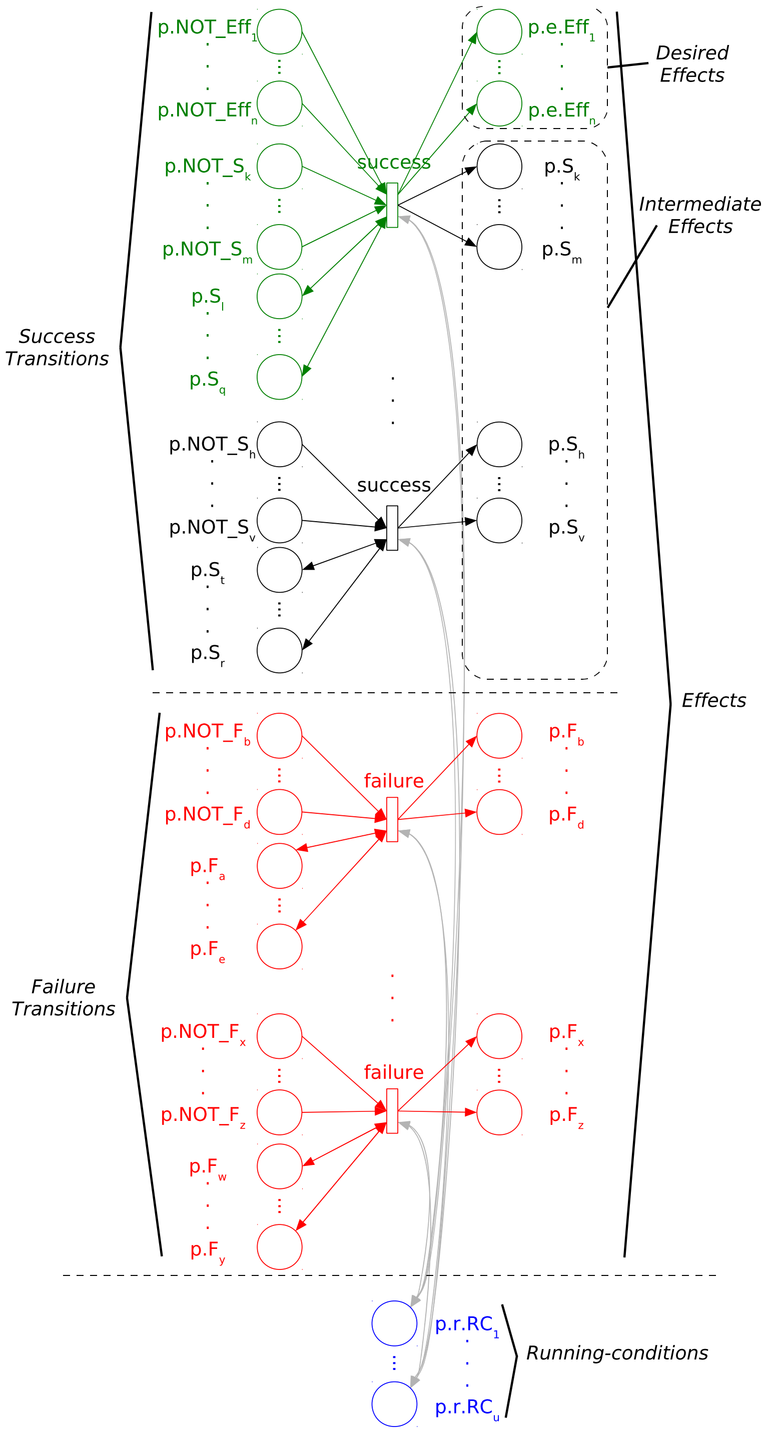
– all places in P_R have "`r.`" after the "`predicate.`" prefix;
– all places in $P_{E_{S_D}}$ have "`e.`" after the "`predicate.`" prefix;
– all transitions t_j in T_S are labelled $\texttt{success}_j$ or $\texttt{s}_j$;
– all transitions t_j in T_F are labelled $\texttt{failure}_j$ or $\texttt{f}_j$.

The fundamental difference from traditional models (e.g. STRIPS (Fikes and Nilsson 1971)), is that one does not model only the *desired-effects* of the action, but also intermediary and failure effects, and taking a probabilistic framework, through the use of stochastic transitions. Furthermore, as explained before, stochastic time can be used to model uncertainty, providing a modelling approach supported on physics-based reasoning which is not available on well-established probabilistic domain description languages such as PPDDL (Younes and Littman 2004).

The *running-conditions* are input places of all transitions to model the fact that the action can only cause any impact on the environment while these conditions are met. Given that all places are predicate places, rule 5 implies that the action model maintains the predicates according to Definition 3, resulting in a safe Petri net (a safe Petri net is one that has at most one token per place for all markings (Murata 1989)).

As an example, consider an action named `CatchBall`, where the purpose of the robot is to catch a ball. In this case, the robot can only catch the ball if it sees it, resulting in one *running-condition*, `SeeBall`. The *desired-effects* of

Fig. 7 General action model



this action would be catching the ball, i.e., getting the predicate HasBall to true. Including *desired-effects*, *failure-effects* and other intermediary effects, results in the Petri net model shown in Fig. 8.

Failures were explicitly included in the CatchBall action model. Given that this model will be composed with the environment model, (some) failures will already be implicitly modelled through environment transitions (which will be enabled concurrently with action transitions). As such, action models should only include failures explicitly when those events are not modelled by the environment model, or if the probability of those events occurring is much higher when the action is being executed, as compared to the event probability modelled by the environment model. Intermediate effects play an important role in maintaining the state of world consistent. For instance, in the example given, it does

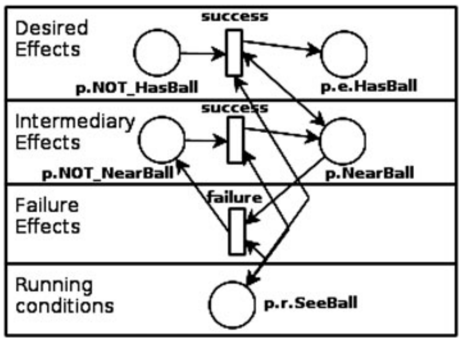


Fig. 8 Petri net model of action `CatchBall`

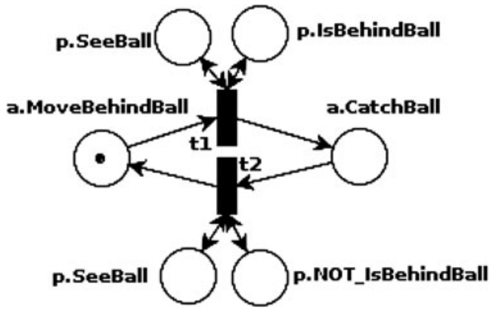


Fig. 9 Petri net model of the `Get_Ball` task plan

not make sense for the robot to gain possession of the ball unless it gets near the ball first.

For execution purposes the Action Executor models are currently used partially, by taking into consideration the *running-conditions* to prevent enabling each action outside of their scope. If the *running-conditions* of a given an action are not enabled that action will not be executed, even if selected by an higher layer.

4.3 Action coordinator layer

The Action Coordinator layer contains Petri net models of robot task plans. A Petri net model of a task plan consists of a MOPN which models predicate-based decisions.

Definition 6 A task plan model is a MOPN where

1. All places $p_j \in P$ are either *predicate places*, *memory places*, *action places* or *task places*.
2. If there is an arc from place p_n, associated to predicate $\mathcal{P}$, to transition t_j, then there is an arc from t_j back to place p_n.

Note that $\mathcal{P}$ is a generic placeholder for a predicate, be it the negative literal or the positive literal. We will now provide the formal definition of action and task places.

Definition 7 An *action place* p_n, labelled "action.`ACTION`" (or, alternatively, prefixed with "*a.*"), is a place associated with action `ACTION`, meaning that if place p_n has at least one token, then action `ACTION` is enabled.

Definition 8 A *task place* p_n, labelled "task.`TASK`" (or, alternatively, prefixed with "*t.*"), is a place associated with task plan `TASK`, meaning that if place p_n has at least one token, then task plan `TASK` is enabled.

Having an action enabled means the world can evolve according to the corresponding Petri net action model, as described in Sect. 4.2. In Definition 6, item 1 implies that a plan can contain task places, which also correspond to task plans, leading to a layered Petri net task plan model without a predefined depth limit. Item 2 is of particular importance, since environment and action models are the only ones which model changes in the world state, while task plans model action/task selection decisions based on those predicates. In task plan models, transitions correspond to predicate based decisions which trigger the execution of actions and/or other task plans. Note that action and task places can only be used in task plan models.

To help understand these definitions, we will follow an example of a soccer playing robot. Consider the task `Get_Ball`, depicted in Fig. 9, which is used for the robot to capture the ball. This plan corresponds to a loop between actions `MoveBehindBall` and `CatchBall`. Note that the initial marking in the task plan models is very important, since it determines which actions, or included tasks, should run when the task is started.

To understand better the concept of tasks within tasks, consider now a task plan for a full soccer-playing robot, depicted in Fig. 10. Here, besides action `Dribble2Goal` and task `Shoot_For_Goal`, one also uses the `Get_Ball` task plan described previously.

5 Petri net expansion for analysis

5.1 Motivation for analysis

Given that all layers are modelled using Petri nets, we can compose all these models together in a single Petri net model. This single Petri net model represents the overall task, which we can analyse a priori. This analysis can cover both logical (e.g. deadlocks) and probabilistic (e.g. probability of reaching a given state) performance properties.

Furthermore, there is a number of properties that must be met during design time, which allow for error detection at an early stage of development. As an example consider the boundedness of the net. Given that we are using predicate places (as defined in Definition 3), these can have only one or zero tokens. If one detects more than one token in a predicate place at design time, or that the sum of tokens in

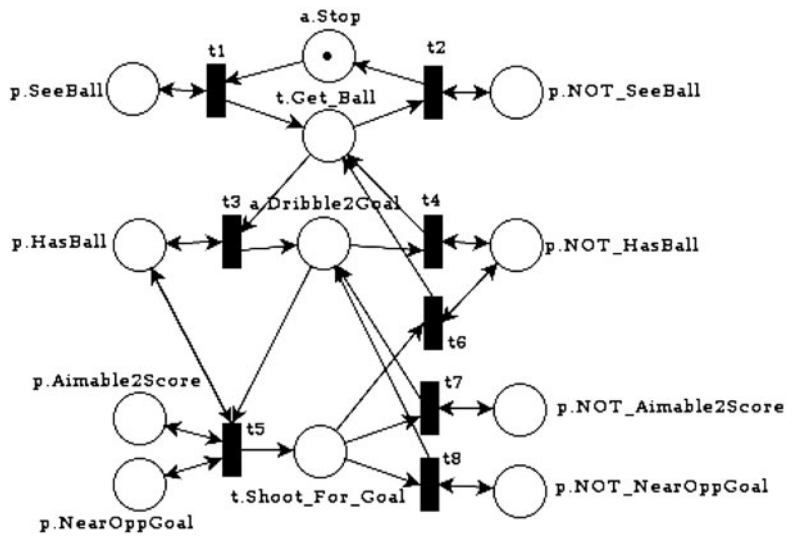


Fig. 10 Petri net model of the `Score_Goal` task plan

the two places associated with a predicate is not always one, it means that there is an error in the models. In the predicate places case, this translates to a simple design rule which states that if a given predicate p is an input place of a transition t, then only one of predicate places NOT_p or p can be an output place of transition t. If additionally one requires action and task places to have at most one token, it results in a safe net requirement. If the total number of tokens in the two places associated with a predicate is constant (equal to one in this case) they form a place invariant (Murata 1989), a property which can be determined from a priori analysis.

Having the modelling and analysis processes integrated under the same framework allows for a design process based on a continuous loop of design-analysis-design. This loop guides the development of the tasks in a structured way, leading to improved task plans even before gathering results from the execution process.

Furthermore, data can also be extracted from the execution process in order to analyse the task a posteriori, and to further improve the models.

5.2 Expansion process

The *Expansion Process* is based on place fusion (Girault and Valk 2003), enabling us to obtain the single Petri net for analysis by merging all the environment, action and task Petri net models. The place labels play an important role in this process, since they allow to distinguish between the different types of places. The current expansion algorithm was designed to work with safe Petri nets, which is our case by construction.

When composing the various Petri nets, there are three basic rules which are always present in the various developed algorithms:

1. If predicate places p_i and p_j are associated with the same predicate, then $p_i = p_j$, i.e., these are duplicate places;
2. Action, task and memory places are always different places, regardless of their label;
3. All transitions are different, regardless of their label.

Regarding Item 1, note that identifying two places associated with the same predicate means identifying two places with the same label.

Given these rules, whenever two Petri nets are composed, if there are several predicate places associated with the same label, then these are duplicate places and can be merged. Merging these places consists on keeping just one of the places while maintaining all the connections of the removed duplicate places.

The following items describe the algorithms developed to obtain this single Petri net from the task, action and environment models. As will be described later, the main guiding principles for the development of these algorithms, was to guarantee that a task and/or action were always interruptible in zero time, and that a task must always start in its initial state (corresponding toy its initial marking).

5.2.1 Petri net complement

Recall that when designing the Petri net models one is not forced to include the two predicate places associated with each predicate. As such, the missing places must be added during the expansion process (e.g., `p.NOT_P` when only `p.P` is present), through the use of Algorithm 1, which we denote *Complement Algorithm*. We will denote $\widehat{PN}$ as the Petri net model which results from complementing Petri net PN.

As an example, applying the complement algorithm to the Petri net model depicted in Fig. 5 results in the model shown in Fig. 11. This is the same model as depicted in Fig. 4, minus the tokens in predicate places `p.NOT_BallMovingSlow` and `p.NOT_BallStopped`

Algorithm 1: Petri net complement algorithm

```
Input: Petri net model, $PN$.
Output: Complemented Petri net model, $\widehat{PN}$.
1  begin
2      Merge all duplicate places in $PN$;
3      foreach place $p_j$ in $PN$ do
4          if $p_j$ is a predicate place then
5              If $\neg p_j$ does not exist yet, add it;
6          else
7              Create a complementary place of $p_j$,
               denoted $\neg p_j$, with marking
               $\#(\neg p_j) = 1 - \#(p_j)$;
8          end
9          foreach input transition $t_i$ of $p_j$ that is not
           an output of $p_j$ or $\neg p_j$ do
10             Add an arc from $\neg p_j$ to $t_i$;
11         end
12         foreach output transition $t_i$ of $p_j$ that is not
           an input of $p_j$ or $\neg p_j$ do
13             Add an arc from $t_i$ to $\neg p_j$;
14         end
15     end
16 end
```

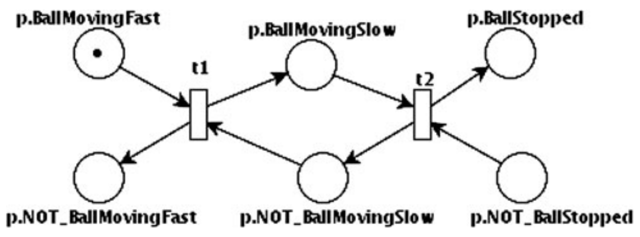


Fig. 11 Complemented Petri net

(the marking of the Petri net is only set during the analysis phase, depending on the initial state of the world).

Note that the complement algorithm also introduces complementary places to non-predicate places. The importance of these complementary places will be more clear later.

5.2.2 *Extended reachability graph*

Before detailing the actual expansion algorithm, one needs to introduce another auxiliary algorithm, used to compute the *Extended Reachability Graph*. The extended reachability graph will allow determining all the possible markings of a Petri net, considering all predicate states. The reachability graph (Murata 1989) of a Petri net with initial marking $\mathcal{M}_0$, denoted $\mathcal{G}(\mathcal{M}_0)$, allows us to determine the reachability set, denoted $\mathcal{R}(\mathcal{M}_0)$, i.e., the set of all reachable markings from the given initial marking.

Definition 9 Given a Petri net model PN of a task plan, the *Extended Petri net model* of that task plan, denoted $\triangle PN$, is obtained by adding transitions to PN such that predicates can switch values in any state. Considering an initial marking $\mathcal{M}_0$, the reachability graph of $\triangle PN$ is the *Extended Reachability Graph* of PN, and is denoted $\triangle\mathcal{G}(\mathcal{M}_0)$. The reachability set of $\triangle PN$ is the *Extended Reachability Set* of PN, and is denoted as $\triangle\mathcal{R}(\mathcal{M}_0)$.

Algorithm 2 describes how to create the extended Petri net models, used to compute the extended reachability graph. Note that, in line 9, all the predicates are set. This is done in order to guarantee that the generated Petri net is in a state were all the predicate places follow Definition 3 and all the added transitions are enabled. This will allow the generation of the Active Reachability Set as described in the next Section. Although we set all positive predicates to 0 and all the negative predicates to 1, we could have done the opposite, without influencing the computation of the Active Reachability Set (see next Section for more details).

Algorithm 2: Petri net extension algorithm

```
Input: Complemented Petri net model, $\widehat{PN}$.
Output: Extended Petri net model, $\triangle\widehat{PN}$.
1  begin
2      Merge all duplicate places in $\widehat{PN}$;
3      foreach place $p_j$ in $\widehat{PN}$ do
4          if $p_j$ is a predicate place then
5              Add a new transition $t'$ with an arc from
               $p_j$ to $t'$ and an arc from $t'$ to $\neg p_j$;
6              Add a new transition $t''$ with an arc from
               $\neg p_j$ to $t''$ and an arc from $t''$ to $p_j$;
7          end
8      end
9      Set all predicate place markings associated with
       positive and negative literals, $p_j$ and $\neg p_j$, to 0 and
       1 respectively;
10 end
```

As an example, the extended Petri net model of task `Get_Ball` (shown in Fig. 9) is depicted in Fig. 12.

5.2.3 *Reduced and active reachability sets*

Definition 10 Given a reachability set $\mathcal{R}(\mathcal{M}_0)$ of task plan Petri net model PN, for an initial marking $\mathcal{M}_0$, the *Reduced Reachability Set* of PN for the given initial marking, denoted $\nabla\mathcal{R}(\mathcal{M}_0)$, is obtained by removing all predicate places from the markings. The marking obtained by removing the predicate places is called *reduced marking* and is denoted as $\nabla\mathcal{M}$.

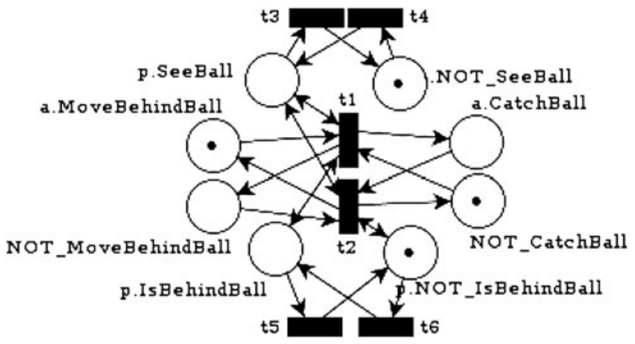


Fig. 12 Extended Petri net model of the `Get_Ball` task plan.

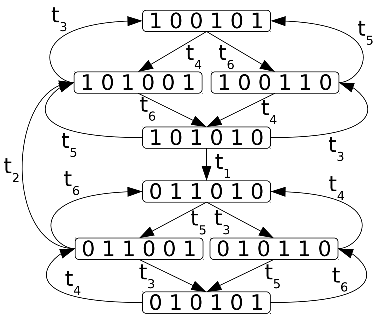


Fig. 13 Extended reachability graph for the `Get_Ball` task plan Petri net model

Note that we have not defined the *Reduced Reachability Graph*, as it is not applicable. As such, although it makes sense to determine $\nabla(\triangle\mathcal{R}(\mathcal{M}_0))$, computing $\triangle(\nabla\mathcal{R}(\mathcal{M}_0))$ is undefined.

Definition 11 Given a task plan Petri net model PN, with an initial marking $\mathcal{M}_0$, the *Active Reachability Set* of PN for the given initial marking, is defined as $\mathfrak{R}(\mathcal{M}_0) \equiv \nabla(\triangle\mathcal{R}(\mathcal{M}_0))$ with $\mathfrak{M}_j$ representing the markings in $\mathfrak{R}(\mathcal{M}_0)$.

The active reachability set represents all the possible states in terms of action and task places (hence the name *active*), regardless of the predicate place markings. Obtaining all possible states for a Petri net in terms of actions and task places, independently of the predicate states, will allow any task plan Petri net model to interrupt/terminate an included task Petri net model, by including exit transitions from all the possible states (regarding action and task places), as explained in the next section.

As an example, consider again the `Get_Ball` task plan in Fig. 9. Computing the active reachability set of this task plan, $\mathfrak{R}(\mathcal{M}_0)$, implies that one needs to determine the extended reachability graph of the Petri net model depicted in Fig. 9, which implies computing the reachability graph of the Petri net model shown in Fig. 12, resulting in the graph presented in Fig. 13 (considering that the marking place labels are given by {`MoveBehindBall`, `CatchBall`, `SeeBall`, `NOT_SeeBall`, `IsBehindBall`, `NOT_IsBehindBall`}).

The extended reachability set of the `Get_Ball` task plan Petri net model is given by all the markings shown in Fig. 13. Computing the reduced reachability set of the extended `Get_Ball` task plan Petri net model, where the marking is given by {`MoveBehindBall`, `CatchBall`} , results in the following active reachability set:

$$\mathfrak{M}_0 = \{1, 0\}, \qquad \mathfrak{M}_1 = \{0, 1\}$$

5.2.4 Petri net expansion

The actual expansion process is performed using Algorithm 3. This algorithm was written considering that the resulting Petri net model must follow these guiding principles:

1. A task should always be started in its initial reduced marking;
2. When a task is not enabled, the associated Petri net model reduced marking must only contain zeros;
3. All tasks and actions should be interruptible in zero time.

Basically the algorithm merges all environment, action and task models, starting from the main top-level task model (t_{Net}). Transitions which lead to the termination of tasks are replicated (one per each low-level state) allowing the lower level tasks to terminate no matter what state they are in, given that we consider the active reachability set.

The action places function as enabling places of all transitions on the associated models, i.e., if there is a token in the action place, then the transitions of the associated Petri net model are enabled (as long as the *running-conditions* and remaining input predicate places are also true). As such, expanding an action place consists of adding arcs from the action place to all the action model transitions and back.

Task places, besides working also as enabling places for the associated Petri net model, must guarantee that the task is always interruptible, and that no action or lower level task keep running when the task is disabled. This is the reason one needs to create complementary places for all places (including non-predicate places), and output transitions for every possible reduced marking of the associated Petri net model.

Prefixing the transitions with the model names during the expansion algorithm enables us to determine to which model they belong when performing the analysis of the final model.

Regarding the complexity of the expansion process, the main bottleneck of the algorithm is in lines 19 to 27 of Algorithm 3, particularly concerning the addition of transitions for each active marking of the added task Petri net. Note that, if the added task Petri net as n states and m output transitions, the algorithm will add $m * (n - 1)$ transitions and respective arcs to the expanded Petri net, resulting in a geometric complexity of the algorithm. Furthermore, given the

Algorithm 3: Single Petri net generation algorithm

```
Input: Environment and top-level task ($t_{Net}$) models
Output: Single, expanded Petri net
1  begin
2      Create an empty Petri Net, denoted $f_{Net}$;
3      foreach environment model, $e_{Net}$ do
4          Add $\widehat{e_{Net}}$ to $f_{Net}$ and merge duplicate places;
5      end
6      Add $\widehat{t_{Net}}$ to $f_{Net}$ and merge duplicate places;
7      foreach task place, then each action, in $f_{Net}$ do
8          Compute $\widehat{m_{Net}}$, the complemented Petri net
           model associated with the place being expanded;
9          Prefix $\widehat{m_{Net}}$ transition labels with $m_{Net}$ name;
10         if the expanding place is an action place then
11             Add $\widehat{m_{Net}}$ to $f_{Net}$;
12         else
13             Compute $\Re(\mathcal{M}_0)$ of $\widehat{m_{Net}}$;
14             Add $\widehat{m_{Net}}$ to $f_{Net}$;
15             foreach input transition $t_i$ of the
               expanding place in $f_{Net}$ do
16                 Add an arc from transition $t_i$ to all
                   places containing one token in $\mathfrak{M}_0$;
17                 Add an arc from all non-predicate
                   $NOT\_p_j$ places to transition $t_i$;
18             end
19             foreach output transition $t_n$ of the
               expanding place in $f_{Net}$ do
20                 foreach $\mathfrak{M}_j$, with $j > 0$ do
21                     Add a new immediate transition
                       $t'_n$ with the same input places and
                       output places as $t_n$;
22                     Add an arc from each place with
                       one token in $\mathfrak{M}_j$ to transition $t'_n$;
23                     Add an arc from transition $t'_n$ to
                       all non-predicate $NOT\_p_j$ places;
24                 end
25                 Add an arc from each place containing
                   one token in $\mathfrak{M}_0$ to transition $t_n$;
26                 Add an arc from transition $t_n$ to all
                   non-predicate $NOT\_p_j$ places;
27             end
28             if the expanding place has zero tokens
               then
29                 Remove the tokens from all places
                   with a token in $\mathfrak{M}_0$ and add a token
                   to all non-predicate $NOT\_p_j$ places;
30             end
31         end
32         Add an arc from the expanding place to all
           transitions in $\widehat{m_{Net}}$;
33         Add an arc from all transitions in $\widehat{m_{Net}}$ to the
           macro place;
34         Merge duplicate places;
35         Prefix the macro place label with an “e” to
           denote that it has been expanded;
36     end
37     Remove the tokens from all predicate places;
38 end
```

modularity of the framework, each task is expected to be small and have a small to medium number of states. It results that the current limitation of the framework in terms of complexity is mainly in the analysis process, as some of the obtained properties are based on the Reachability Graph generated from the full Petri net. The generation of the full Petri net takes a time geometrically proportional to the number of places, but the analysis time can grow exponentially with the number of states.

5.3 Example

In order to illustrate the application of the algorithms described in the previous sections, we will use a simplified example of a soccer playing robot. The process is depicted in Fig. 14, with the environment, action and task plan models shown in the top left corner. The expansion process follows the algorithm described in Algorithm 3, with the various steps illustrated in Fig. 14 and described in the following list (the numbering used here is the same numbering used in the arrows shown in Fig. 14, while the line numbers correspond to the ones in Algorithm 3):

1. Create an empty model, add all environment models and merge duplicate places (lines 2 to 5);
2. Add complemented `Play_Ball` task model (line 6);
3. Merge duplicate places (line 6);
4. Expand task place `task.Get_Ball` by:
   - 4.1. Computing the active reachability set of the complemented `Get_Ball` task plan model, resulting in two active markings: marking 0, with one token in places {`eaction.FindBall` and `NOT_Catch-Ball`}; and marking 1, with one token in places {`NOT_FindBall`, `eaction.CatchBall`} (lines 8 to 13);
   - 4.2. Adding complemented `Get_Ball` task plan model (line 14);
   - 4.3. Add input transition arcs, allowing task `Get_Ball` to start in its initial marking when enabled through a transition to place `task.Get_Ball` (lines 15 to 18);
   - 4.4. Add output transition arcs, enabling task `Get_-Ball` to be interrupted/ended when the task is in its:
     - 4.4.1. Active marking 1 (lines 20 to 24);
     - 4.4.2. Active marking 0 (lines 25 to 26);
   - 4.5. Add enabling arcs, so that each transition of task `Get_Ball` plan model can only fire when task place `task.Get_ Ball` has a token (lines 32 to 33);
   - 4.6. Merge duplicate places (line 34) and prefix the macro place with “e” (line 35);
5. Expand action `FindBall` by adding the complemented action model (lines 8 to 11), enabling arcs (lines 32

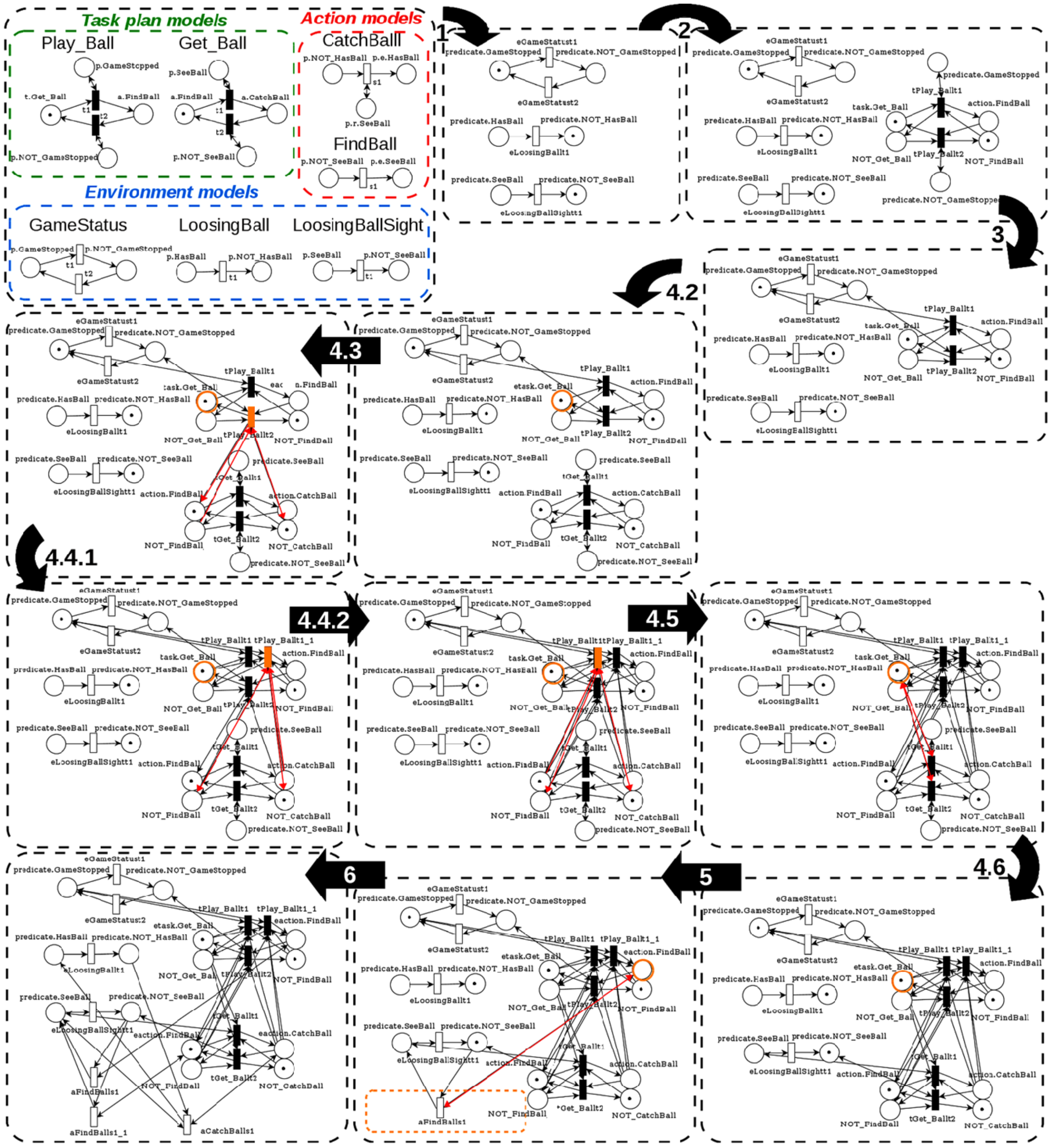


Fig. 14 Expansion example, with several steps illustrated

to 33), merging duplicate places (line 34) and prefixing the macro place with "e" (line 35);

6. Expand remaining actions (same lines as previous item) and remove all predicate place tokens (line 37), resulting in the final Petri net.

After having obtained the single Petri net, one needs to choose an initial state for the environment by setting the number of tokens in each predicate place. This Petri net can then be analysed using well known techniques (Murata 1989; Viswanadham and Narahari 1992; Bause and Kritzinger 2002), for properties such as the probability that a condition holds, the amount of time spent in each action and/or task, or the throughput of each transition, among others.

6 Models identification

The environment and action models can be created manually. However, in many situations, this might prove to be a cumbersome experience, especially if the number of actions and/or predicates increases or if one intends to have a more realistic model. As such, a method was developed to create these models automatically from real data, based on the following steps:

1. Specify a list of actions with their *running-conditions* and *desired-effects*;
2. Specify, if needed, experiment episode *start state* and *end state*;
3. Run data collection experiment;
4. Estimate the action and environment models.

Several techniques exist in the literature to obtain the structure and parameters of the models in model-based techniques (Dupont et al. 2005), depending on the problem. Our method is similar to the one used with Probabilistic Deterministic Finite Automata, however, instead of focusing on the language generated by the produced events, we focus on the states and determining the transition parameters. Although, for each state, we do not know a priori the set of reachable states, given that each state is completely defined by the full set of predicate values, we can deterministically identify which state the robot estimates it is in (whether the state was correctly observed or not), having an upper bound of the number of states which can be found. As such, and as described in more detail later, the structure can be obtained directly from the predicate values and their changes. As stated in Dupont et al. (2005), estimating the state transitions parameters can then be achieved using the appropriate maximum likelihood estimator according to the probability distribution. Finally, we translate the obtained models into Petri net models in order to use them in the context of the proposed framework. The identification is done on a per action basis, following the modular nature of the framework, which further allows us to easily switch between different environment models and evaluate the performance impact of that change.

As an example, we will consider throughout this section a scenario with one robots and one action, `CatchBall`. In this case, we start by defining action `CatchBall` *running-conditions*, predicate `SeeBall`, and *desired-effects*, predicate `HasBall`.

6.1 Data collection experiment

The data collection experiment is used to gather information about the actions impact in the world in order to estimate the action and environment models, particular which and when transitions occur. The experiment consists on running each

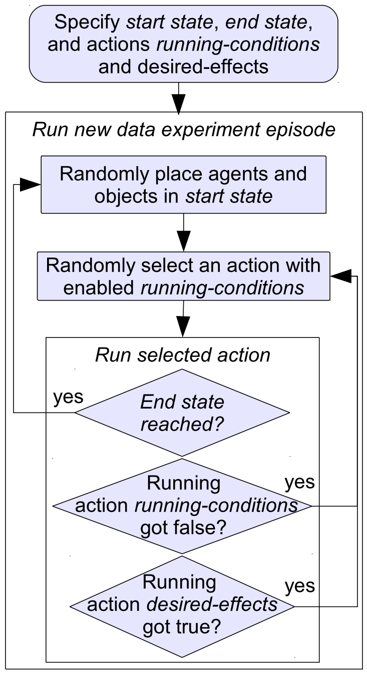

Specify *start state*, *end state*, and actions *running-conditions* and desired-effects
Run new data experiment episode
Randomly place agents and objects in *start state*
Randomly select an action with enabled *running-conditions*
Run selected action
yes
End state reached?
Running action *running-conditions* got false?
yes
Running action *desired-effects* got true?
yes

Fig. 15 Models identification data collection experiment

action based on the actions *running-conditions* and *desired-effects*. The experiment flowchart is depicted in Fig. 15.

The user starts by specifying predicate values that describe the *start* and *end* states, and both the *running-conditions* and *desired-effects* for each action. The *start* predicates describe the conditions that need to be met for an experiment episode to be able to start, while the *end* predicates describe the conditions that force a new experiment episode to be started. Note that both can be empty, which means that any state can be used as a starting state, and that there is no pre-condition for an episode to end (in this case an episode ends by user termination or by specifying an amount of experiment time). When a new episode is started, all the agents and objects in the environment are randomly placed so that the *start* predicates are met. Then, one action is randomly selected from the subset of actions with *running-conditions* met, and started. This action runs as long as the *end* state predicates are not met, the action *running-conditions* are met, and the *desired-effects* are not met. If the *end* state predicates are met, a new episode is started and, if the *running-conditions* are not met, or the *desired-effects* are met, a new action is selected. Note that when performing this selection due to the current action *desired-effects* becoming true, the same action can be selected again.

In our case, we also included a counter per action to hold the number of times each action was selected during all data collection experiment episodes. These counters are used both to determine when the experiment should end and to increase the selection probability of less selected actions.

Continuing the example used in this section, the *start* conditions could be an empty set, while the *end* conditions could be: `HasBall` OR `NOT_SeeBall`. This means that if one of these predicates gets true, a new episode is started, with the ball and robot placed randomly.

6.2 Model estimation

Once the data collection is accomplished, the goal is to obtain the Petri net model of each action and of the environment from the experiment data, such that the analysis can be performed using automatically obtained realistic models. For each episode ran during the data collection experiment we have all predicate values and changes, plus the selected action over time. Every time one or more predicates changes, we have a state change. Since one action is enabled at a time, each state change can be attributed either to the enabled action or the environment (environment transitions occur in parallel with the action transitions). Given enough time, we will eventually capture all possible state changes, however, given that the a priori knowledge consists only of the available predicates list, and the actions *running-conditions* and *desired-effects*, we cannot know if a state change is caused by the action or the environment, or both. As such, and since during the data collection experiment we are associating each state change with an action, each identified action model will in fact (partially) contain the environment model, i.e., part (or all) of the environment model will be embedded in each action model. Details on how to overcome this limitation will be given later, in Sect. 9.

Having processed all data we obtain a Markov chain model for each action (plus environment), where each state is defined by the entire set of predicate values. Each event is defined by the set of predicates that are changed and the set of predicates that keep their value, i.e., the event is fully defined by the states it connects. We can create a GSPN model containing only stochastic transitions, associated to the events, whose marking process is equivalent to the obtained Markov chain.

Continuing with the example followed throughout this section, consider, for instance, that the experiment first episode started in state {`SeeBall`, `NearBall`, `NOT_HasBall`} . In this case, given that the `CatchBall` *running-conditions* are verified, the action is started. Eventually, after some time, changes will occur in the world. For instance, the robot might catch the ball (event e_1), thus switching to state {`SeeBall`, `NearBall`, `HasBall`}. This state matches the *end* conditions, thus a new episode will be started, by repositioning the robot and ball randomly. Considering that this new state is {`SeeBall`, `NOT_NearBall`, `NOT_HasBall`}, the action `CatchBall` is once again started. Eventually, after some time, changes will occur in the world. For instance, the robot

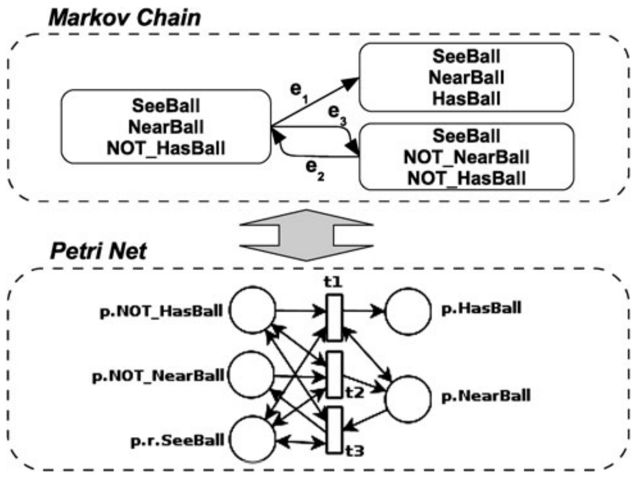


Fig. 16 Hypothetical identified Petri net model

might have gotten closer to the ball (event e_2), yielding state {`SeeBall`, `NearBall`, `NOT_HasBall`}. Continuing with the action execution, the robot might stop being close to the ball (event e_3), yielding state {`SeeBall`, `NOT_NearBall`, `NOT_HasBall`}. This results in the Markov chain depicted in Fig. 16, which is translated to a Petri net by considering each event a transition, as described earlier. As can be seen, all identified transitions have as inputs the predicate places associated with predicates that were true in their input state, and have as outputs predicate places associated with all predicates which were true in the output state.

For the GSPN model to be fully specified, one needs to compute the transition rates. From the experiment data, we have for each event, for each action, a set of elapsed times corresponding to the time taken from the start state to the reached state on every occurrence of the event. With this information one can compute the transition rates for event e_i for a state-transition pair. Since we consider all transitions to follow an exponential distribution, the probability of the event e_i occurring in a given state s, for a given action a, is given by Murata (1989):

$$ {}^{a}_{s}P(e_i) = \frac{{}^{a}_{s}\lambda_i}{\sum_{j=1}^{{}^{a}_{s}N} {}^{a}_{s}\lambda_j}, \tag{2} $$

where ${}^{a}_{s}N$ is the number of events that can occur in state s, in action a.

For each state, for each action, the probability of an event occurring can be estimated based on its frequency, considering all occurrences of all events that can occur in that state:

$$ {}^{a}_{s}P(e_i) = \frac{\#^{a}_{s}\Delta t_{e_i}}{\sum_{j=1}^{{}^{a}_{s}N} \#^{a}_{s}\Delta t_{e_j}}, \tag{3} $$

where ${}^{a}_{s}\Delta t_{e_i}$ is a vector with the elapsed time obtained for each occurrence of event e_i in state s, in action a, and # denotes vector length, i.e., the number of occurrences.

Since the time for each event to occur follows an exponential distribution, and we consider all events to be independent, the rate at which any event occurs in a given state, in a given action, is the sum of the rates of all events occurring in that state. Furthermore, the maximum likelihood estimator of the rate parameter for an exponential distribution is given by the inverse of the mean of the elapsed times, resulting in

$$ {}^{a}_{s}\lambda = \sum_{j=1}^{{}^{a}_{s}N} {}^{a}_{s}\lambda_j = \left(\frac{1}{\sum_{i=1}^{{}^{a}_{s}N} \#^{a}_{s}\Delta t_{e_i}} \sum_{l=1}^{{}^{a}_{s}N} {}^{a}_{s}\Delta t_{e_l} \right)^{-1} \quad (4) $$

It results that we can estimate the rate of each event using:

$$ \begin{aligned} {}^{a}_{s}\lambda_i &= {}^{a}_{s}P(e_i) * \sum_{j=1}^{{}^{a}_{s}N} {}^{a}_{s}\lambda_j \\ &= \frac{\#^{a}_{s}\Delta t_{e_i}}{\sum_{j=1}^{{}^{a}_{s}N} \#^{a}_{s}\Delta t_{e_j}} * \left(\frac{1}{\sum_{i=1}^{{}^{a}_{s}N} \#^{a}_{s}\Delta t_{e_i}} \sum_{l=1}^{{}^{a}_{s}N} {}^{a}_{s}\Delta t_{e_l} \right)^{-1} \\ &= \frac{\#^{a}_{s}\Delta t_{e_i}}{\sum_{l=1}^{{}^{a}_{s}N} {}^{a}_{s}\Delta t_{e_l}} \end{aligned} $$

The rate associated with the event is the rate associated with the corresponding transition.

In practise we will not have an infinite number of episodes, which implies that one might not capture all existing transitions. Since higher probability transitions are captured with higher probability, the impact on the performance properties should be negligible, given an enough number of episodes. However, there can be a higher impact on logical properties, particularly concerning reachable states. Nevertheless, most of the logical properties are still valid. For instance, both the invariant properties over predicate places and the safeness of the net must hold (for the states found), even if not all transitions were identified.

Since we cannot know if we have detected all possible transitions, currently the number of episodes is selected empirically. Nevertheless we are studying measures that might give us a run-time indication of how well our approximation is doing, enabling us to automatically determine when an experiment should end.

7 Execution of single-robot tasks

Task plans developed using our modelling framework are executed using a Petri net execution application, which is part of our Multiple Robot Middleware for Intelligent Decision-Making (MeRMaID) (Barbosa et al. 2007). MeRMaID allows using the same code both in the real robots and in the simulator, and provides monitoring algorithms which update predicate values by processing sensor data (e.g., whether an object is present in the image or not). Given a Petri net based task plan model, the *Petri net Executor* checks which transitions are enabled, considering the currently selected actions and true predicates, and fires them accordingly. The token flow will eventually enable and trigger the execution of new actions. An action will stop execution if it achieves its *desired conditions* or if any of its *running-conditions* becomes false.

When a task plan is executed, time associated to transition firing will depend on the time until the *desired conditions* become true or any of the *running-conditions* becomes false. When analysing a model quantitatively, the estimated environment models include the estimated firing rates, and the closed-loop Petri net properties are determined based on Markov chain equations or through simulation. Task execution can be monitored in order to log experimental results. Those can be compared later to the theoretical results, obtained from model analysis.

8 Results

In order to test the developed algorithms, several experiments were performed using the WeBots (Michel 1998) simulator, comparing the results based on Markov chain based Petri net analysis with the experimental results (obtained by running the tasks in the simulated robots). When using real robots one should act similarly, though the results would then come from real and not simulated experiments. All results were obtained on a 1.86 GHz dual-core Intel Pentium processor with 3 Gb of memory, with the test scenario consisting of:

20 predicates: `NearOwnGoal`, `MidField`, `NearOppGoal`, `HasBall`, `IsBehindBall`, `NearBall`, `SeeBall`, `BallKicked`, `RobotStopped`, `GoalOpportunity`, `Aimable2Score`, `BallOwnGoal`, `BallNearOwnGoal`, `BallMidField`, `BallNearOppGoal`, `BallOppGoal`, `BallMovingToOwnGoal`, `BallMovingToOppGoal`, `BallMovingFast`, `BallStopped`;

1 experimenting robot with 6 actions: `Aim2Score`, `CatchBall`, `Dribble2Goal`, `Kick2Goal`, `MoveBehindBall` and `Stop`[1];

9 robots running `Move2StartPosition` action for obstacles.

[1]Since we considered that the robot could always see the ball, the `Stop` action is not actually used in the experiments, so we will not include results concerning this action.

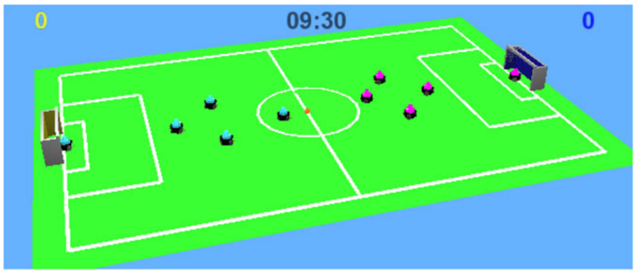


Fig. 17 Overview of the experiment world in the WeBots simulator

An overview of the test scenario is given in Fig. 17. Since there are no predicates concerning other robots, and since the 9 non-experimenting robots are always trying to stay in the same position, these can be seen as static obstacles. We considered that the robot could always see the ball, meaning that predicate `SeeBall` was always true.[2]

8.1 Identification results

For the data collection experiment the *start state* was undefined, and the *end state* was defined as having predicates `BallOwnGoal` or `BallOppGoal` true. Furthermore, whenever the ball enters the goal, it stays in the goal until a new episode is started. As such, no identified model will include transitions having simultaneously input predicate `BallOwnGoal` and output predicate `NOT_BallOwnGoal`, or input predicate `BallOppGoal` and output predicate `NOT_BallOppGoal`, i.e., there will be no identified transition modelling the removal of the ball from the goal.

When comparing the theoretical results with the experimental ones, we will consider 4 different data collection experiments, A, B, C and D, corresponding to a data collection experiment with enough episodes to run each action at least 10, 100, 1000 and 10 000 times respectively. Note that experiment A data is contained in experiment B data, which is contained in C, which is contained in D.

While running each experiment, and whenever a transition is fired, we update the total number of transitions fired for the selected action and, if the transition was never fired before, the number of distinct transitions found is increased. In Figs. 18a and 18b you find the plot of the number of distinct transitions fired versus the total number of transitions fired, for experiment D. Although the ratio between the number of distinct transitions fired and the total number of transitions fired is decreasing with the number of episodes, the number of new transitions found is still increasing by a reasonable rate. This means the system contains a very large number of transitions and that, even after running each action 10 000 times, there are still transitions and states that were never identified. We will see later that this has a limited impact on the task analysis results.

[2]Although predicate `SeeBall` was always true, we opted to keep it in order to minimise the differences to the task plans running in the real robots.

Table 1 Normalised number of distinct transitions fired per action (D)

Exp.	A2S	CB	D2G	K2G	MBB
A	3.32	2.34	6.94	1.3	3.93
B	3.31	2.32	7.84	1.96	4.51
C	2.95	2.3	8.21	2.02	4.77
D	2.98	2.22	8.15	2.05	5.01

The larger number of transitions fired for some actions (see Fig. 18a) is not only a result of that action enabling an higher number of transitions, but also due to the conditions associated with each action. For instance, in our test scenario, action `Move2Ball` *running-conditions* only includes `SeeBall`, meaning that it could always be selected during the data collection experiment (given that we considered predicate `SeeBall` to be always true), while `Dribble2Goal` *running-conditions* include predicates `SeeBall` and `HasBall`. As such, even with the weights included to increase the selection probability of less selected actions, some actions can only be selected in a much smaller state space, leading to a smaller number of selection times over time.

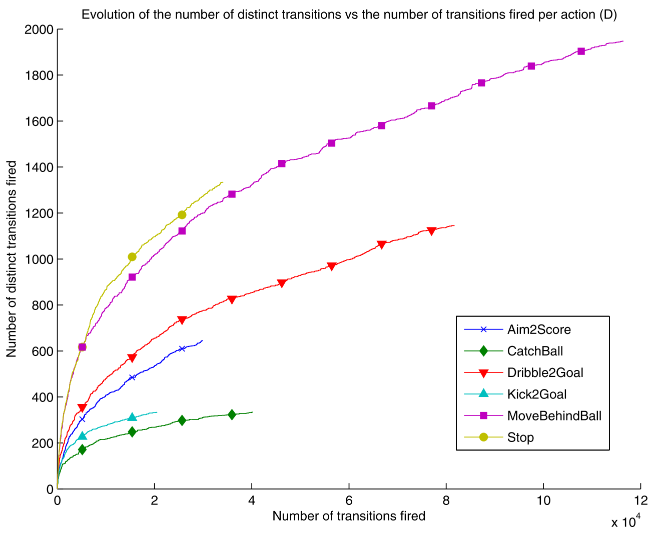
To get a better comparison of the number of distinct transitions per action, Table 1 shows the number of distinct transitions fired per action divided by the total number of times that action was selected (the action names were shortened to decrease space). These numbers give us an indication of the relative amount of data needed per action and could eventually be used to determine which actions should need a larger number of data collection episodes. For instance, 5.01 for `MoveBehindBall` action in data set D, means that, in average, each time this action was selected it lead to the occurrence of 5.01 different transitions.

8.2 Stochastic performance analysis

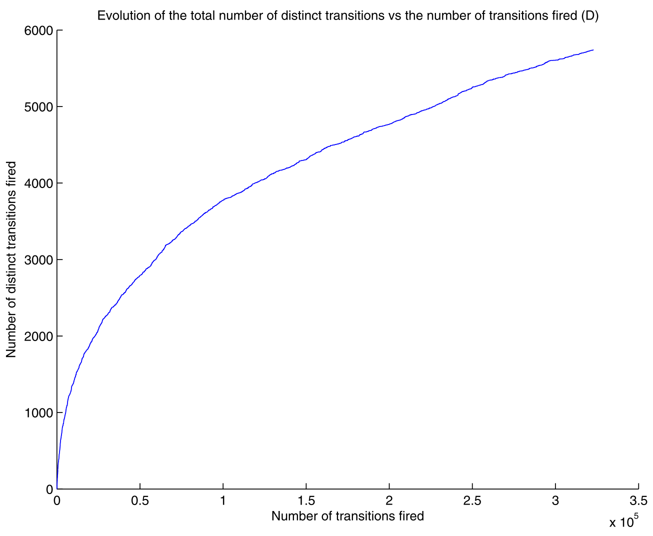
In order to compare the theoretical results with the experimental ones, we devised three different tasks based on the task shown in Fig. 10, with slight modifications:

`Score_Goal_Shoot_First` Whenever the robot captures the ball (predicate `HasBall` is true), if the direct path to the opponent goal is free (predicate `Aimable2Score` is true), the robot tries to score immediately;

`Score_Goal_Shoot_50_50` This task is halfway between the other two tasks. Whenever the robots captures the ball, it will randomly decide (with 0.5 probability) if it should try to score or dribble the ball closer to the goal;



(a) Number of distinct transitions fired vs total number of transitions fired, per action.



(b) Total number of distinct transitions fired vs total number of transitions fired

Fig. 18 Number of different fired transitions vs number of fired transitions during identification

`Score_Goal_Shoot_Later` The robot only tries to score if, besides having a direct path to the goal free, it is near the opponent goal (predicate `NearOppGoal` true).

For each of these three manually created task plans, the model (for analysis purposes) is automatically obtained by the expansion process described earlier, generating a single Petri net. In this case, the action and environment models used are those obtained through the identification process described in the previous section. Since the ball is not removed from the goals during each episode, and given that the purpose of these tasks is to score a goal, it is expected that the obtained Petri net model of the overall task will contain deadlocks, corresponding to a goal scored in one of the goals.

We performed a transient analysis of the task and compared it to the transient results of the experiment. In these tests we considered the experimenting robot and ball to be initially positioned near the robot own goal and middle field, respectively.

8.2.1 Transient analysis

We performed the transient analysis of the probability of scoring a goal in the opponent goal for the Petri net generated from `Score_Goal_Shoot_Later`, for each data collection experiment set, obtaining the results depicted in Fig. 19. As can be seen from the figure, the result obtained with 1000 episodes is very similar to the one obtained with 10 000 episodes, despite the number of new transitions still being found (see Fig. 18).

For the experimental results, we ran 10 000 episodes for each task plan in the WeBots simulator, i.e., the experiment ran until the robot scored 10 000 goals (including goals scored both in its goal and the opponent goal). Each episode consisted in placing the robot in the same initial conditions as in the theoretical case and running the task until a goal was scored (the same end conditions as the theoretical transient analysis). Note that in the experimental tests only task plan models and the actions implementation on the robot are used, i.e., no action or environment models are used.

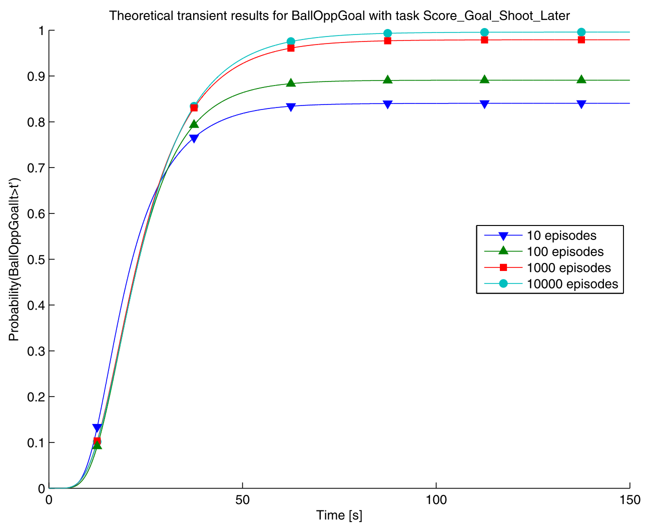
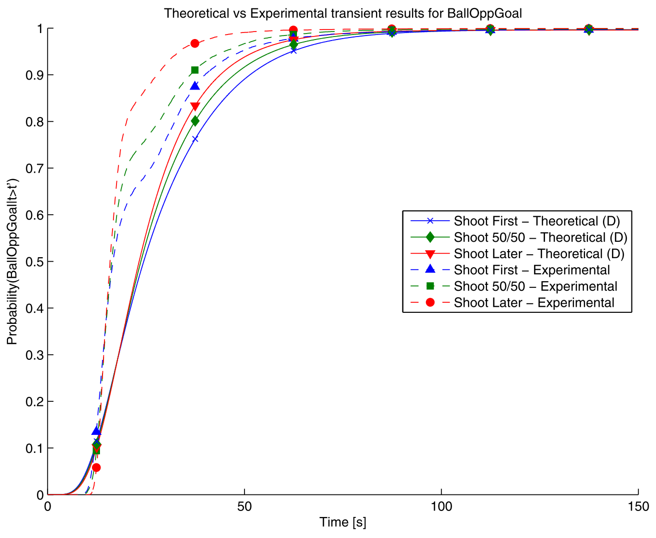
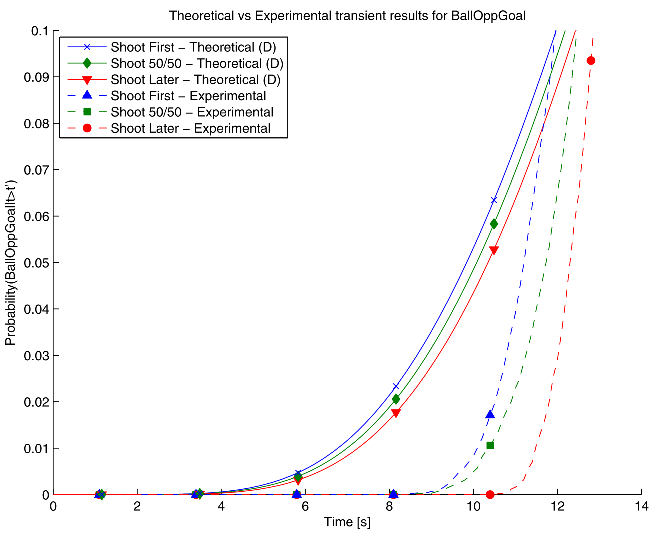


Fig. 19 Theoretical transient analysis for different number of episodes

In Fig. 20a we show the theoretical transient analysis results, obtained using the D data set, versus the experimental results. In spite of all the approximations made, the results display a small error. Furthermore, the relative transient trends of the 3 tasks for the experimental case are similar to



(a) Full time transient analysis results.



(b) Initial time transient analysis results.

Fig. 20 Theoretical vs experimental transient analysis for predicate `BallOppGoal` with experiment D, zooming in the initial time period on the right plot

those of the theoretical case. This is important because one of the goals of the analysis is to give insight on the relative performance of the different tasks.

Analysing each task during the initial 14 s (see Fig. 20b), one can see that shooting earlier leads to higher probability of scoring in the short term, but in the long term it leads to a lower scoring probability. This is expected as when kicking immediately the robot can score sooner, but with a higher failure probability, as opposed to kicking only after getting close to the goal, which takes longer time but has higher probability of scoring. Note that there is a reasonable time difference regarding the increase of scoring probability between the theoretical and experimental result, which is explained by the fact that we are considering that all stochastic timed transitions follow an exponential distribution. In reality, some transitions always have a minimum time which must elapse before the transition can fire, which is not captured by our model. We are currently working on this as future work.

These results give us insight into the task properties, allowing to make decisions which can improve the task performance. For instance, during a match, one could choose to kick immediately, or only when closer to the goal, based on the amount of time missing and the current score.

The comparison of the probability of scoring in our own goal for the theoretical prediction versus the experimental results is depicted in Fig. 21. In all cases the probability of scoring in our own goal is very low and, in both the `Score_Goal_Shoot_50_50` and `Score_Goal_-Shoot_First` tasks, no own goal was scored in 10000 episodes. Nevertheless, the maximum obtained error between the theoretical probability and the real probability is 3.2E–04 (see Fig. 21).

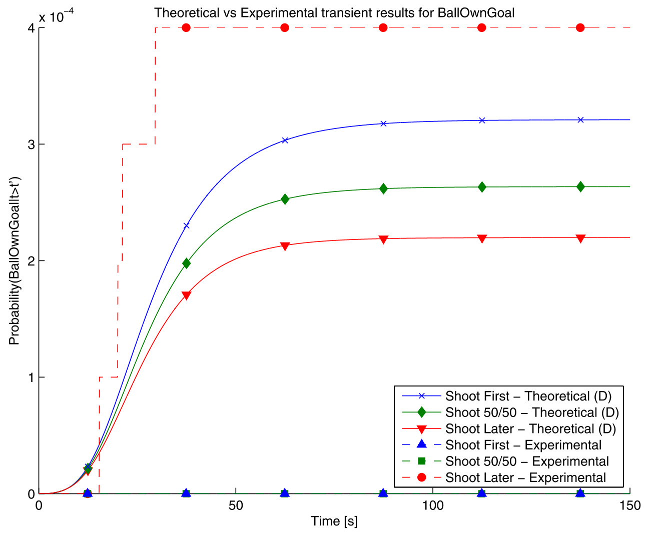


Fig. 21 Theoretical vs experimental transient analysis for predicate BallOwnGoal

An important aspect of these results, is that the sum of the probabilities of predicates `BallOwnGoal` and `Bal-lOppGoal` for the theoretical case when a steady-state is reached is not 1, as shown in Table 2. This result is due to the fact that the generated single Petri net model of the overall task includes deadlocks which do not correspond to a goal scored, either due to not having performed enough data collection episodes for the action and environment models, or due to the fact that the task plan was actually poorly designed. Knowing which of the two answers is the right one

Table 2 Theoretical steady-state probability of scoring goals (from transient analysis). Here, BOWG, BOPG and NG are short terms for `BallOwnGoal`, `BallOppGoal` and "No goal", respectively

Task	Probability		
	BOWG	BOPG	NG
Shoot_First	3.209E–04	9.967E–01	2.978E–03
Shoot_50_50	2.635E–04	9.964E–01	3.359E–03
ShootLater	2.198E–04	9.958E–01	3.958E–03

can only be achieved by analysing the deadlocks found, the path that led to those deadlocks, and the designed task. For the given test cases, we concluded that these unexpected deadlocks were due to not having performed enough data collection episodes. This means that the theoretical Petri net led the world to a state that was never visited by the selected action during the identification process. This conclusion is further backed up by the results shown in Fig. 18 and by the fact that the experimental results did not include any deadlock other than a goal scored.

Given that the final Petri net is safe (by construction, and confirmed by our analysis), all pairs of predicate places form an invariant, and the set of all action places also form an invariant, the average number of tokens per place represents the average time spent in any action, for action places, and the average time a predicate had a given boolean value, for predicate places (for additional performance measures computation check, for instance, Viswanadham and Narahari (1992)).

8.2.2 *Steady-state analysis*

Based on the above results, and in order to further study the impact of the unexpected deadlocks described earlier, we performed a steady-state analysis. With the three task plans shown, and by repositioning the ball in the centre of the field whenever a goal is scored, the robot can play indefinitely. To model and analyse this setup under our framework, all we need to do is add an environment model (manually designed), which models the repositioning of the ball when a goal is scored. For the action models we will use exactly the same data as used in the results above, but will only consider only the D case. The initial state consisted on placing the robot and ball in the field centre.

With this new setup one would expect that the resulting generated Petri net would yield no deadlocks. However, and as explained earlier, since there were not enough data collection episodes, that is not the case. Nevertheless, by performing an analysis of the communication classes (Jarvis and Shier 1999; Ocasio 2009) found, we determined that for each task plan there was one large transient class containing around 95% of the tangible states, and all the remaining classes contained only one absorbing state. As such, we opt

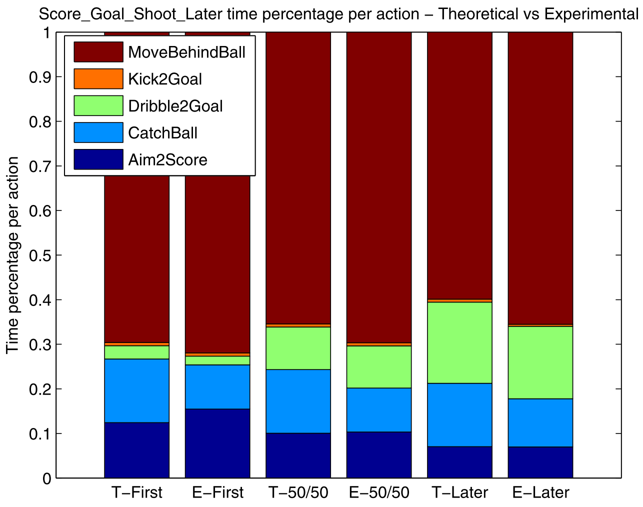


Fig. 22 Theoretical vs experimental average time spent per action

Table 3 Average steady-state error of the number of tokens per place

Tasks	Action places	Predicate places
Shoot_First	2.155E–02	7.346E–02
Shoot_50_50	1.809E–02	7.149E–02
Shoot_Later	2.251E–02	5.128E–02

to perform the steady-state analysis of the task excluding all absorbing states, i.e., considering only the large communication class.

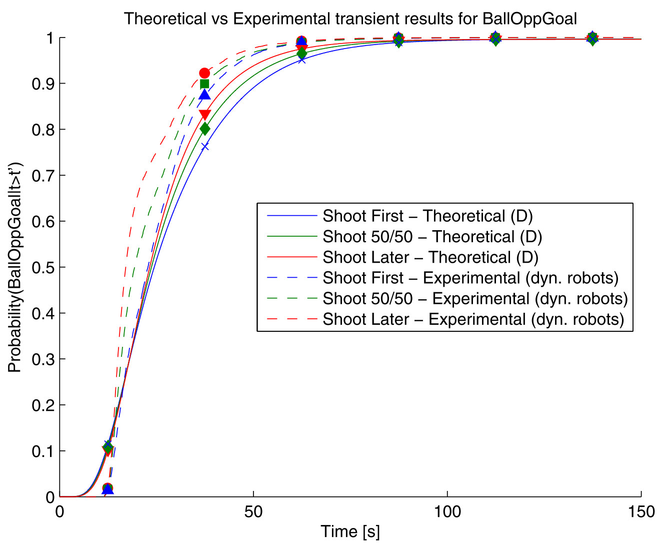
For the experimental results, we let the robot play until the computed average number of predicates per place stabilised, i.e., until we reached an experimental steady-state. For the steady-state analysis we compared the average time spent by the robot in each action, yielding the results shown in Fig. 22.

The steady-state average error, measured as the average absolute difference between the experimental and predicted number of tokens for all action places and for all predicate places is shown in Table 3. This result further strengths the conclusions described earlier, allowing us to use these techniques to perform stationary analysis even in the presence of deadlocks due to insufficient data collection episodes.

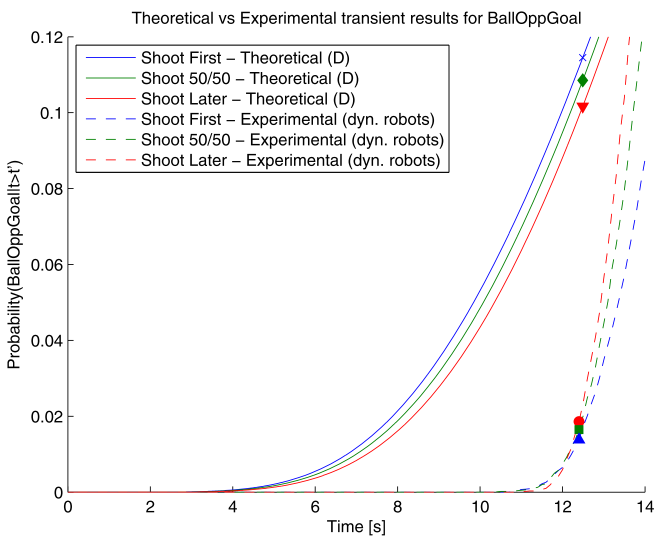
The steady-state analysis also gives us further insight into the task execution. For instance, Fig. 22 shows that the robot spends almost no time in the `Dribble2Goal` when kicking earlier, as opposed to when kicking closer to the goal, as expected. On the other hand, and also as expected, the robot spends more time chasing the ball when kicking earlier.

8.3 Stochastic performance analysis (dynamic robots case)

In order to further evaluate the framework performance, we re-ran the experiments with all the 9 non-experimenting robots navigating randomly around their start posture,



(a) Full time transient analysis results with dynamic robots.



(b) Initial time transient analysis results.

Fig. 23 Theoretical vs experimental (with dynamic robots) transient analysis for predicate `BallOppGoal` with experiment D, zooming in the initial time period on the right plot

with a radius of 1 m and the same reference speed as the experimenting robot. We then compared the outcome of these experiments to the same theoretical analysis performed previously, both for the transient and stationary analysis results, as detailed in the following sections.

8.3.1 Transient analysis

Having the other robots navigating randomly has a negative impact in the transient scoring probability, as can be seen by the shift to the right of the experimental transient result plots, depicted in Fig. 23. As can be seen by comparing Figs. 23a to 20a, the transient results got closer to the theoretic ones in the long run, given that the theoretic analysis estimates a longer time for the same scoring probability. The impact on the results is greater in the initial time period. Given that the experimenting robot is starting in its own field, and given the dynamic movement of the other robots, it is more difficult to score when further away from the opponents goal, as shown in Fig. 23b. This lead to a smaller difference between the various tasks for this initial time period, meaning that the advantage of shooting earlier when time is short is not so large when the other robots are dynamic. In this case, the estimate is worse when compared with the result plotted in Fig. 20b.

8.3.2 Steady-state analysis

In terms of the stationary probability the impact of considering dynamic robots was small, as can be seen by comparing the results in Fig. 24 and Table 4 with the results shown previously in Table 3 and Fig. 22.

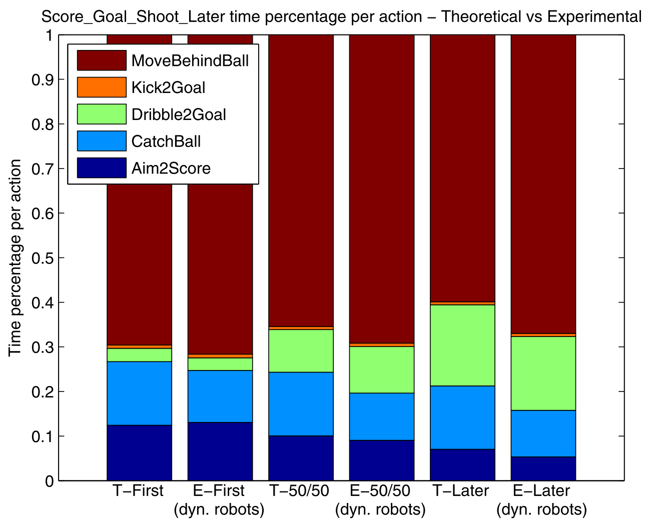


Fig. 24 Theoretical vs experimental (with dynamic robots) average time spent per action

Table 4 Average steady-state error of the number of tokens per place with dynamic robots

Tasks	Action places	Predicate places
Shoot_First	1.111E–02	4.107E–02
Shoot_50_50	1.870E–02	4.995E–02
Shoot_Later	2.851E–02	5.851E–02

Finally, although the theoretical analysis results were based in scenarios with static robots, we can still draw the same conclusions as in the previous experiments: shooting first as a higher probability of scoring sooner but a lower probability of scoring in the long run.

9 Conclusions and future work

This paper introduced an integrated framework for modelling, analysis and execution of robot tasks based on Petri nets. The framework allows modelling both controllable events, representing decisions made by the robot, and uncontrollable events performed by other robots or that result from the environment physics, through the inclusion of environment, action and task models. Furthermore, these models can be created separately and later composed, leading to an easier design of the models, and allowing at the same time to perform analysis of the task as a single integrated model. Petri net representation of plans has modelling capabilities smaller than other richer representations, e.g., predicate calculus based planners. Nevertheless, they bring the advantage of enabling both quantitative and qualitative plan analysis and representation of uncertainty. We plan to work on automatic planners that use the Petri net representation for plans. Two types of analysis can be performed: qualitative (logical) and quantitative (performance), although our emphasis was on the quantitative case. Furthermore, having the action and environment models specified, task analysis is carried out in seconds, as opposed to running simulations, or tests with real robots, which would take hours or days.

The framework provides a design-analysis-design approach, which leads to improved task plans even before having run the task in the real robots. The action models can be designed manually or can be identified through an unmanned data collection experiment. In the latter case, manually designed models can be added to comply with setup changes without having to re-run the data collection experiment. Currently we are not yet capable of identifying the action and environment models separately. We are working on implementing that feature in the near future, either by performing additional analysis on the data collected, or/and by allowing human designed environment models to increase our a priori knowledge of the models. We further included results using a realistic simulator, WeBots, which showed the applicability of the framework to realistic scenarios.

As part of our future work we plan to improve the data collection algorithm by including more active exploration techniques which could provide better decisions of which actions data to collect more often, under which conditions, and which stopping conditions to use. Furthermore, we plan to allow updating the models in real-time during execution, so as to continuously improve the estimated models. Modelling transitions with a minimum time plus exponentially distributed time, instead of only an exponentially distributed time, should also lead to improved results. We are also working on performing experiments in the real robots. The main issue in that case, is supervising and conducting the experiment execution for identification. One approach here is to obtain a small data set to start with, and keep updating the models as the robot executes the tasks themselves. One can even use the simulator model as a starting point, it if is realistic enough, and proceed with the online update.

The task plan models of the proposed examples were manually built, although several work exists in the literature concerning the synthesis of Petri net based controllers from specifications (Hickmott et al. 2007; Dideban and Alla 2008; Mauser and Lorenz 2009). In our work, random switches model concurrency, particularly regarding decision making. By associating action triggering to these transitions, one is effectively modelling a Markov Decision Process (Cassandras and Lafortune 2008; Puterman 1994), whereas the weights associated to the random switch transitions might be computed to meet desired specifications. Here we have the advantage that the state space has already been restricted by the Petri net structure. As future work we plan to integrate these synthesis techniques into the framework. Furthermore, the information included regarding actions, namely the running-conditions and desired effects, can also be used in the automatic creation of plans.

We have already introduced some basic multi-robot modelling capabilities in the framework (Costelha and Lima 2008), by using communication actions with predicates representing the result of those communications, but yet without containing selection mechanisms. Several approaches exist for this purpose, including Petri net-based (Toktam Ebadi and Purvis 2009), and we are currently working on integrating them in our framework.

Acknowledgements This work was supported by the Portuguese Fundação para a Ciência e Tecnologia under grant SFRH/BD/12707/2003 and ISR/IST pluriannual funding through the PIDDAC Program funds.

References

Akin, H. L., Birk, A., Bonarini, A., Kraetzschmar, G., Lima, P., Nardi, D., Pagello, E., Reggiani, M., Saffiotti, A., Sanfeliu, A., & Spaan, M. (2008) White paper on network robot systems, and formal models and methods for cooperation. http://aass.oru.se/Agora/EuronCoop/Docs/SIG_CoopRob_WhitePaper.pdf.

Barbosa, M., Ramos, N., & Lima, P. (2007). MeRMaID - multiple-robot middleware for intelligent decision-making. In *6th IFAC symposium on intelligent autonomous vehicles IAV2007* Amsterdam: Elsevier.

Basu, A., Gallien, M., Lesire, C., Nguyen, T. H., Bensalem, S., Ingrand, F., & Sifakis, J. (2008). Incremental component-based construction and verification of a robotic system. In *Proc. of the 2008 European conference on artificial intelligence (ECAI 2008)* (24–29).

Bause, F., & Kritzinger, P. S. (2002). *Stochastic Petri nets: an introduction to the theory* (2nd edn.). Berlin: Vieweg Verlag.

Bernardinello, L., & Cindio, F. D. (1992). A survey of basic net models and modular net classes. In *Advances in Petri nets 1992, the DEMON project* (304–351). Berlin: Springer.

Cassandras, C. G., & Lafortune, S. (2008). *Introduction to discrete event systems* (2nd edn.). New York: Springer.

Castelnuovo, A., Ferrarini, L., & Piroddi, L. (2007). An incremental petri net-based approach to the modeling of production sequences in manufacturing systems. *IEEE Transactions on Automation Science and Engineering*, *4*(3), 424–434.

Costelha, H., & Lima, P. (2008). Modelling, analysis and execution of multi-robot tasks using petri nets. In *Proceedings of the 7th international joint conference on autonomous agents and multiagent systems, international foundation for autonomous agents and multiagent systems (AAMAS '08)* (pp. 1187–1190).

Dideban, A., & Alla, H. (2008). Reduction of constraints for controller synthesis based on safe Petri nets. *Automatica*, *7*(7), 1697–1706.

Dupont, P., Denis, F., & Esposito, Y. (2005). Links between probabilistic automata and hidden Markov models: probability distributions, learning models and induction algorithms. *Pattern Recognition*, *38*(9), 1349–1371.

Espiau, B., Kapellos, K., Jourdan, M., & Simon, D. (1995). On the validation of robotics control systems. Part I. High level specification and formal verification. Tech. rep. 2719, INRIA.

Fikes, R., & Nilsson, N. J. (1971). STRIPS: a new approach to the application of theorem proving. *Artificial Intelligence*, *2*(3), 189–208.

Girault, C., & Valk, R. (2003). *Petri nets for systems engineering*. Berlin: Springer.

Herrero-Perez, D., & Martinez-Barbera, H. (2008). Petri Nets based coordination of flexible autonomous guided vehicles in flexible manufacturing systems. In *IEEE int. conf. on emerging technologies and factory automation (ETFA 2008)* (pp. 508–515).

Hickmott, S., Rintanen, J., Thiébaux, S., & White, L. (2007). Planning via petri net unfolding. In *Proc. of the 20th int. joint conf. on artificial intelligence (IJCAI'07)* (pp. 1904–1911). San Mateo: Morgan Kaufmann.

Jarvis, J., & Shier, D. (1999). Graph-theoretic analysis of finite Markov chains. In *Applied mathematical modeling: a multidisciplinary approach* Boca Raton: CRC Press.

Kim, G., & Chung, W. (2007). Navigation behavior selection using generalized stochastic Petri nets for a service robot. *IEEE Transactions on Systems, Man and Cybernetics. Part C, Applications and Reviews*, *37*(4), 494–503.

King, J., Pretty, R., & Gosine, R. (2003) Coordinated execution of tasks in a multiagent environment. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 615–619.

Košecká, J., Christensen, H. I., & Bajcsy, R. (1997). Experiments in behaviour composition. *Robotics and Autonomous Systems*, *19*(3–4), 287–298.

Kress-Gazit, H., Fainekos, G. E., & Pappas, G. J. (2009). Temporal logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, *25*(6), 1370–1381.

Mauser, S., & Lorenz, R. (2009). Variants of the language based synthesis problem for Petri nets. In *Ninth int. conf. on application of concurrency to system design* (pp. 89–98).

Michel, O. (1998) Webots—fast prototyping and simulation of mobile robots.

Murata, T. (1989). Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, *4*, 541–580.

Ocasio, VA (2009). Stability of boolean dynamical systems and graph periodicity. Master's thesis, University of Puerto Rico, Mayagüez Campus.

Petri, C. A. (1966). Kommunikation mit automaten. Tech. rep., Bonn: Institut für Instrumentelle Mathematik, english translation.

Puterman, M. L. (1994). *Markov decision processes—discrete stochastic dynamic programming* (1st edn.). New York: Wiley.

Qin, Y., & Xu, R. (2009). GSPN-based modeling and analysis for robotized assembly system. In *IEEE int. conf. on robotics and biomimetics, 2008 (ROBIO 2008)* (pp. 1070–1075).

Röck, A., & Kresman, R. (2006). On Petri nets and predicate-transition nets. In *Proceedings of the international conference on software engineering research and practice & conference on programming languages and compilers, (SERP 2006)* (pp. 903–909).

Toktam Ebadi, M. P., & Purvis, M. (2009) A framework for facilitating cooperation in multi-agent systems. The Journal of Supercomputing.

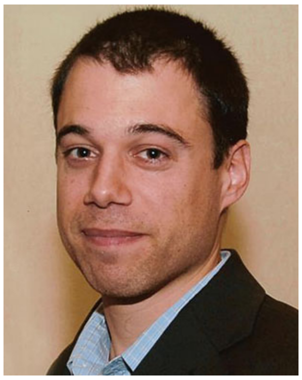
Viswanadham, N., & Narahari, Y. (1992). *Performance modeling of automated manufacturing systems*. New York: Prentice Hall.

Wang, F., Kyriakopoulos, K., Tsolkas, A., & Saridis, G. (1991). A Petri-net coordination model for an intelligent mobile robot. *IEEE Transactions on Systems Science and Cybernetics*, *21*(4), 777–789.

Younes, H. L. S. & Littman, M. L. (2004). PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Tech. rep. CMU-CS-04-167, Carnegie Mellon University.

Zhou, M., & Venkatesh, K. (1999). *Modeling, simulation and control of flexible manufacturing systems*. Singapore: World Scientific Publishing.

Ziparo, VA, & Iocchi, L. (2006). Petri net plans. In *Proc. of the fourth int. workshop on modelling of objects, components, and agents (MOCA'06)* (pp. 267–290). Hamburg: University of Hamburg.



Hugo Costelha got his Ph.D. (2010) in Electrical Engineering at the Instituto Superior Técnico, Lisbon Technical University, Lisbon Portugal. Currently, he is an Assistant Professor at School of Technology and Management, Polytechnic Institute of Leiria. He is also a researcher at the Institute for Systems and Robotics, a Portuguese private research institution, in the Intelligent Robots and Systems group. His research interests include the areas of discrete event systems applied to multi-robot systems, field robotics and educational robotics. Hugo Costelha is a member of the IEEE and the Portuguese Robotics Society (SPR), where he is currently presiding the Service Robotics Committee. He has participated in numerous robotics events, including RoboCup and the Portuguese Robotic Open since 2002. He has been very active in the promotion of Science and Technology through robotics, having been part of the Organizing Committee of the 2010 and 2011 editions of the Portuguese Robotics Open, being the vice-chair of the 10th Conference on Mobile Robots and Competitions held together with the 2010 edition.

Pedro Lima got his Ph.D. (1994) in Electrical Engineering at the Rensselaer Polytechnic Institute, Troy, NY, USA. Currently, he is an Associate Professor at Instituto Superior Técnico, Lisbon Technical University. He is also a member of the Institute for Systems and Robotics, a Portuguese private research institution, where he is coordinator of the Intelligent Robots and Systems group. His research interests lie in the areas of discrete event systems and decision-making under uncertainty, namely their applications to multi-robot systems. Pedro Lima is a Trustee of the RoboCup Federation, and was the General Chair of RoboCup2004, held in Lisbon. He was the President of the Portuguese Robotics Society (2009–2011), and is a senior member of the IEEE. He is the co-author of two books, member of the Editorial Board for the journal Robotics and Autonomous Systems (Elsevier) and regularly serves as member of international conferences program committees, and has coordinated national (FCT, AdI) and international (ESA, EU) R&D projects. He has also been very active in the promotion of Science and Technology to the society, through the organization of Robotics events in Portugal, including the Portuguese Robotics Open since 2001. He spent a sabbatical leave at the Universidad Carlos III de Madrid, Spain, where he was awarded a 6-month Chair of Excellence by the University Board of Governors.