4

# Petri Net Robotic Task Plan Representation: Modelling, Analysis and Execution

Hugo Costelha
*Institute for Systems and Robotics, Instituto Superior Técnico*
*Superior School of Technology and Management, Polytechnic Institute of Leiria*
*Portugal*

Pedro Lima
*Institute for Systems and Robotics, Instituto Superior Técnico*
*Portugal*

## 1. Introduction

As the usage of robots in everyday tasks increases, there is a need to improve our knowledge concerning the execution of those robotic tasks. Robotic task models are usually not based on formal approaches but tailored to the task at hand. Applying discrete event system concepts to model robotic 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.

Most of the work found on the literature concerning the design of robotic tasks using Discrete Event Systems is based on Finite State Automata for code generation (Dominguez-Brito et al., 2000), qualitative specifications (Kosecka et al., 1997), some quantitative specifications (Espiau et al., 1995), modularisation (Kosecka et al., 1997) and even to model multi-robot systems (Damas & Lima, 2004). Work using Petri nets to design robotic tasks under temporal requirements, focusing also on the generation of real-time, error-free code can be found in (Montano et al., 2000). Petri net Plans were introduced in (Ziparo & Iocchi, 2006) for design and execution of task plans. However, these do not close the loop, i.e., do not consider the actual implications of the actions on the environment, focusing mostly on the design.

In this chapter we describe a Petri net based framework which allows a systematic approach for modelling, analysis and execution of robotic tasks. This framework is divided in three layers: task plan models, action models and environment models. The models range from the robot decision-making algorithms (task plan models) to the environment dynamics, due to physics and/or actions of other agents (environment models).

In the proposed models, Petri net places represent tasks, primitive actions and logic predicates set by sensor readings. These logic predicates provide and abstraction of the world relevant features. By composing these models, and applying analysis techniques, important a priori information can be obtained regarding the properties of the task. The models are based on Marked Ordinary Petri Nets and Generalised Stochastic Petri Nets (Murata, 1989), allowing

for transitions to be immediate or stochastic, and leading to both the retrieval of logical properties, such as deadlocks and resource conservation, and (probabilistic) performance properties, such as probability or average time to reach a desired state.
Given the action and environment models, different task plans can be quickly evaluated using the analysis techniques, allowing for a priori quality/performance based decisions. Furthermore, due to the introduced abstractions and inherent Petri net restrictions, the state space is reduced.
By introducing comunication models we further extend the framework to model cooperative robotic tasks, namely those involving the coordination of two or more robots, which require the exchange of synchronisation messages, either using explicit (e.g., wireless) or implicit (e.g., vision-based observation of teammates) communication. Different communication models allow the analysis of different scenarios, such as perfect communication, delayed communication or absence of communication.
Extensive tests were done using a robotic soccer scenario under full observability.

## 2. Petri Nets

Petri nets (Petri, 1966) are a widely used formalism for modelling Discrete Event Dynamic Systems (DEDS). 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 robotic tasks.
Petri nets are preferred to Finite State Automata (FSA) due to their larger modelling power and because one can model the same state space with a smaller graph. Moreover, although composition of Petri nets usually leads to an exponential growth in the state space (as for FSA), graphically the growth is linear in the size of the composed graphs given that the state is distributed. This makes the design process simpler for the task designer, and helps managing the display of the tasks both for monitoring and designing purposes. Moreover, we use Marked Ordinary Petri Nets (MOPN) and Generalised Stochastic Petri Nets (GSPN) (Murata, 1989), allowing the retrieval of logical and (probabilistic) performance properties.
Modularity in Petri nets is achieved since each resource can be modeled separately and then composed with others. Although composition operators exist for FSA, Petri nets can model subsystems with input and output places, so that they can be connected as in a circuit.

### 2.1 Marked Ordinary Petri Nets

The simplest models we use are Marked Ordinary Petri nets:

**Definition 2.1.** *A marked ordinary Petri net is a five-tuple* $PN = \langle P, T, I, O, \mathcal{M}_0 \rangle$*, where:*

- $P = \{p_1, p_2, \ldots, p_n\}$ *is a finite, not empty, set of places;*
- $T = \{t_1, t_2, \ldots, t_m\}$ *is a finite set of transitions;*
- $I = P \times T$ *represents the arc connections from places to transitions, such that* $i_{lj} = 1$ *if, and only if, there is an arc from* $p_l$ *to* $t_j$*, and* $i_{lj} = 0$ *otherwise;*
- $O = T \times P$ *represent the arc connections from transition to places, such that* $o_{lj} = 1$ *if, and only if, there is an arc from* $t_l$ *to* $p_j$*, and* $o_{lj} = 0$ *otherwise;*
- $\mathcal{M}(j) = [m_1(j), \ldots, m_n(j)]$ *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.*

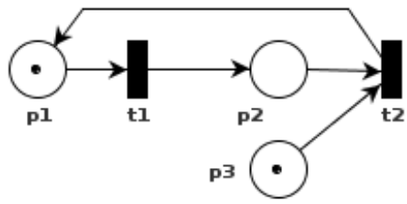



Fig. 1. A simple Petri net.

A simple MOPN is depicted in Fig. 1. Basically we have two types of nodes: places, represented by circles, and transitions, represented by filled rectangles. The places can contain any number of tokens, represented by the number of dots (or a number) inside the place. For instance, in the Petri net shown in Fig. 1 place $p_1$ and place $p_3$ both have one token, while place $p_2$ has zero tokens.
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. For instance, the initial marking of the Petri net from Fig. 1 is given by $\mathcal{M}_0 = [1, 0, 1]$.
In this class of Petri nets, all the transitions are *immediate* (have zero firing time), i.e., once they are enabled and fired, the new marking is instantly reached.
When referring to input or output nodes of a particular node, we are referring to the nodes connected to or from that node. For instance, transition $t_3$ has places $p_2$ and $p_3$ as its input places, while it has only one output place, $p_1$.

### 2.2 Generalised Stochastic Petri Nets

MOPNs are suited for qualitative analysis, but not for performance analysis. For this purpose, one uses generalised stochastic Petri nets.

**Definition 2.2.** *A standard GSPN is an eight-tuple* $PN = \langle P, T, I, O, \mathcal{M}_0, R, S \rangle$, *where:*

- $P, T, I, O, \mathcal{M}_0$ *are as defined in 2.1;*
- $T$ *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,* $R\left(t_{E_j}\right) = \mu_j$, *where* $\mu_j$ *is called the firing rate of* $t_{E_j}$;
- $S$ *is a set of* random switches, *which associate probability distributions to subsets of conflicting immediate transitions.*

Stochastic (exponential) transitions, once enabled, fire only when an exponentially distributed time $d_j$ has elapsed. This definition of GSPNs includes also the possibility of associating a probability distribution to conflicting immediate transitions, by the use of the *random switches*. These random switches can be static (invariant to the marking of the net) or dynamic (dependent on the marking of the net).
We use a particular implementation of random switches, by associating weights to the immediate transitions, as described in Definition 2.3.

**Definition 2.3.** *A GSPN is an eight-tuple* $PN = \langle P, T, I, O, \mathcal{M}_0, R, W \rangle$, *where:*

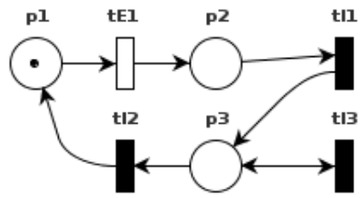



Fig. 2. Generalised stochastic Petri net.

- *$P, T, I, O, \mathcal{M}_0, R$ are as defined in 2.2;*
- *$W$ is a function from the immediate transitions set $T_I$ to a set of real numbers, $W\left(t_{I_j}\right) = w_j$, where $w_j$ is the weight associated with immediate transition $t_{I_j}$;*
- *For any given marking, the probability of firing an enabled transition $t_i$ is equal to $w_i/\mathcal{W}$, where $\mathcal{W}$ is the sum of the weights of all enabled transitions for the given marking.*

Consider the GSPN model depicted in Fig. 2. In this example, $t_{E_1}$ is an exponential timed transition (drawn with a unfilled rectangle), while $t_{I_1}$, $t_{I_2}$ and $t_{I_3}$ are immediate transitions with associated weights. Initially $t_1$ is enabled, since $p_1$ has tokens, and will fire after an exponentially distributed time with rate $\mu_1$ 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 conflicting transitions, whereas only one will fire, according to the following probabilities:

$$P_f\left(t_{I_2}\right) = \frac{w_2}{w_2 + w_3} \qquad P_f\left(t_{I_3}\right) = \frac{w_3}{w_2 + w_3}$$

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 & Narahari, 1992). A Markov chain can be obtained from the marking process, and the transition probability matrix computed by using the firing rates of the exponential timed transitions and the probabilities associated with the random switches. This enables the use of tools already available to analyse Markov chains directly from the GSPN, instead of relying on, e.g., Monte Carlo simulation.

### 2.3 Additional Specifications

In our framework, we embody the Petri net models with some additional building blocks, namely macro places, and make use of the place labels to distinguish between different types of places, such as: *action macro places*, *predicate places* and *regular* (or *memory*) *places*. These different types of places do not introduce any change regarding the execution of the Petri nets, but are key in the analysis process explained later.
Regular (or memory) places are normal Petri net places, without any special properties. The remaining types of places are described in the following sections.


Fig. 3. Representation of predicate by a set of places.

### 2.3.1 Predicate Places

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

**Definition 2.4.** *A predicate place $p$ is a place associated with the predicate $P$, described by $p \models P$, such that:*

- $\forall_j, P_j = true \Leftrightarrow m_p(j) = 1$
- $\forall_j, P_j = false \Leftrightarrow m_p(j) = 0$,

*where $P_j$ is the predicate $P$ at time step $j$.*

Basically, a place representing a predicate has one token if that predicate is true and zero tokens otherwise.

**Definition 2.5.** *A Petri net model of a predicate is a MOPN where:*

- $P = \{\neg p, p\}$, *where and $\neg p$ and $p$ are predicate places associated with predicates $\neg P()$ and $P()$ respectively;*
- $I = \emptyset$;
- $O = \emptyset$;
- $\forall_j \mathcal{M}_j = [0,1] \vee [1,0]$.

Although we could achieve the same results by using just one place for representing a predicate, that would lead to the use of inhibitor arcs. Once again we rather maintain the use of the base Petri nets, with minimal extensions added, so as to be able to use a larger set of available Petri tools. Furthemore, although it increases the number of places, it does not increase the state space, and provides a cleaner interface to the user.
As an example, a Petri net model representing the predicate `SeeBall` is depicted in Figure 3. 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.

### 2.3.2 Macro Places

Macros, albeit not always using the same definition, are used to create hierarchical Petri nets (Bernardinello & Cindio, 1992), leading to a higher degree of modularity. The use of macro places allows the drawing of entire Petri net models from lower layers has single places in higher layers, providing for cleaner and reusable models.
Places associated with macros will also have a particular prefix in the place label. Furthermore, since macro places represent entire Petri nets, we need to have expansion algorithms when obtaining one single Petri net without macro places. These details will be given later in Section 4.1.

## 3. Modelling Single-Robot Tasks using Petri Nets

The base framework used throughout this work was developed aiming at:

**Modularity** - fostering the reuse of developed components;

**Design** - providing an intuitive, and possibly graphical, task design solution;

**Analysis** - providing means to analyse a robotic task both before and after its execution;

**Execution** - keeping the models suitable for execution, taking into account that its implementation would have to follow the framework theoretical foundations.

To achieve these goals, a Petri net based solution was developed, using four different layers, as depicted in Figure 4.

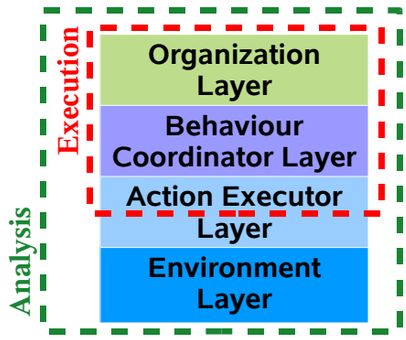



Fig. 4. Models Hierarchy.

Each layer is formed by a set of Petri net models which represent different granularity levels, being the Environment layer the bottom one, and the Organisation layer the top one. The meaning of each layer is as follows:

**Environment Layer** Petri net models at this level represent changes made by other agents (such as other robots) or even physics (such as the braking of a free rolling ball);

**Action Executor Layer** At this level we find Petri net models of the actions, representing the changes performed in the environment by these actions, and the conditions under which these changes can occur;

**Action Coordinator Layer** Here lies the Petri net based task plan models, which basically consist of compositions of actions;

**Organisation Layer** This layer is where higher decision models appear, such as goal selection, thus consisting of compositions of Action Coordinator Layer models.

As can be seen in Figure 4, all models are used in the analysis process, but only the two higher layers and, partially, the Action Executor layer models will be used for execution. This will be further explained in the following sections. Note that, currently, we have not implemented the Organisation layer yet.

### 3.1 Environment Layer

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. To model this process using a GSPN model under our framework, we must first discretise it, such that we can describe it through the use of logic predicates. In this example, we could

consider that the ball could be moving fast, slowly or be stopped, and that the ball will, with time, pass from the fastest movement to the stopped state. With this discretisation, we can model the free ball movement with the Petri net model depicted in Figure 5.

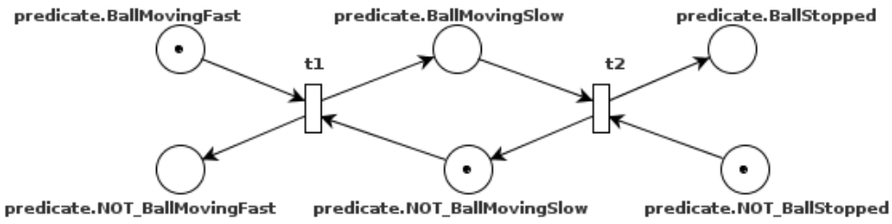



Fig. 5. Petri net model of a moving ball.

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 Figure 6. In this example, the rate at which the ball slows down depends on the weather conditions.

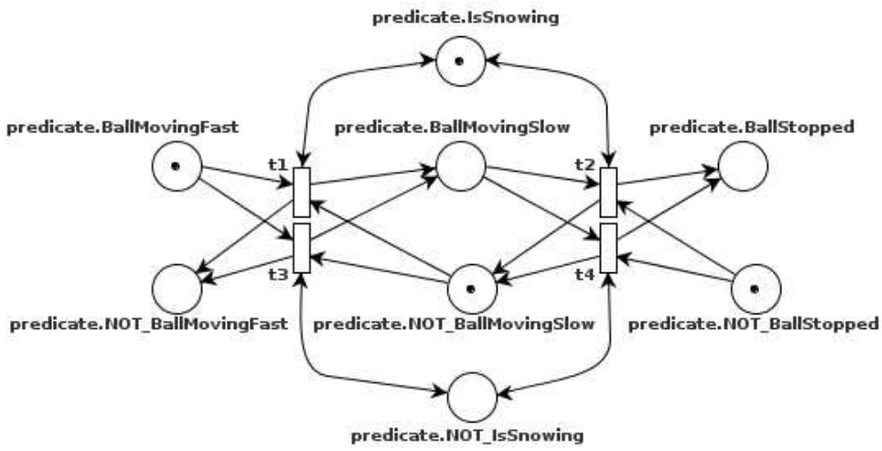



Fig. 6. Petri net model of a moving ball considering thee weather conditions.

### 3.2 Action Executor Layer

Each action Petri net model is a GSPN which represents how the action impacts the environment and under which conditions. As such, each action model consists on a set of transitions representing the environment changes, which can be associated to the success or failure of the action, following the rules described in Definition 3.1. The general model of an action is depicted in Figure 7.

**Definition 3.1.** *A Petri net model of an action is a GSPN, where:*

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

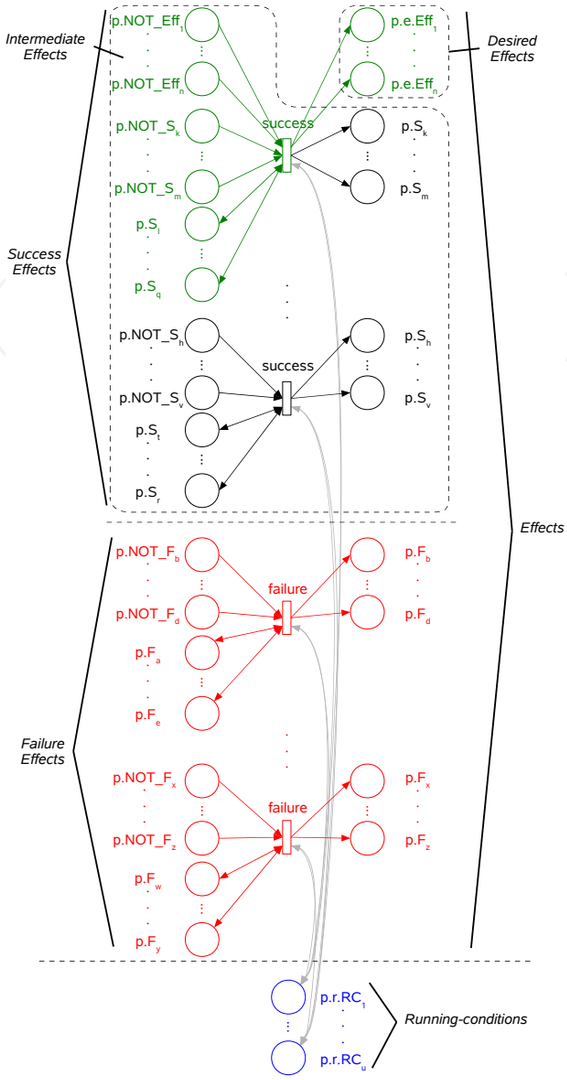



Fig. 7. General action model.

$P_E$ *is the effects place set;*

$P_R$ *is the running-conditions place set;*

2. *All places in* $P_R$ *have "*`r.`*" after the "*`predicate.`*" prefix;*
3. $P_E = P_{E_S} \cup P_{E_F}$*, where* $P_{E_S}$ *and* $P_{E_F}$ *are designated respectively* success places set *and* failure places set.
4. $P_{E_S} = P_{E_{S_I}} \cup P_{E_{S_D}}$*, where* $P_{E_{S_I}}$ *and* $P_{E_{S_D}}$ *are designated respectively* intermediate effects place set *and* desired effects place set.
5. *All places in* $P_{E_{S_D}}$ *have "*`e.`*" after the "*`predicate.`*" prefix;*
6. $T = T_S \cup T_F$ *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;*

7. *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 $p_n$;*
8. *All transitions have one input arc from each running-condition;*
9. *If a desired effect place is an output place of a transition, then all the desired effects places are also output places of that transition;*
10. *All transitions $t_j$ in $T_S$ have the label* `success`$_j$ *or* `s`$_j$*;*
11. *All transitions $t_j$ in $T_F$ have the label* `failure`$_j$ *or* `f`$_j$*;*

Having the *running-conditions* as input places of all transitions models the fact that the action can only cause any impact on the environment if these conditions are met. Given that all places are predicate places, rule 7 implies that the action model maintains the predicates Definition 2.5, resulting in a safe Petri net (has at most one token per place for all markings).
As an example, consider an action named `CatchBall`, where the purpose of the robot is to catch a ball. It would be expected that the robot could only catch the ball if it were near the ball and if it could see the ball, meaning its *running-conditions* would be `CloseToBall` and `SeeBall`. Furthermore, the *desired-effects* of this action would be catching the ball, i.e., getting the predicate `HasBall` to true. This results in the Petri net model show in Figure 8.

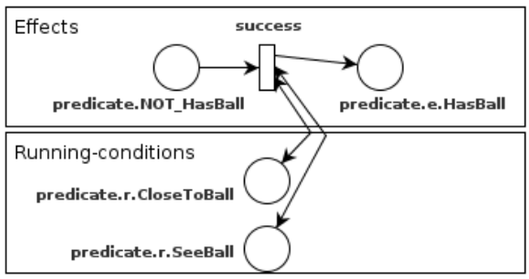



Fig. 8. Petri net model of action `CatchBall`.

Failures were not explicitely included in this model. Although including them is possible, and even expected in many situations, these are already implicitely present, since this model will be composed with the environment model, which models changes performed by others.
For execution purposes the Action Executor models are used partially, by using only the *running-conditions* and *desired-effects* to prevent using each action outside their scope.

### 3.3 Action Coordinator Layer

The Action Coordinator layer contains Petri net models of the task plans. A Petri net model of a task plan consists of a MOPN where places are associated with actions. Places associated with actions are referred to as *action places*, and correspond to action macro places.
To better explain this topic, we will follow an example of a soccer playing robot. In this example, the robot uses actions `Move2Ball`, `CatchBall`, `Dribble2Goal`, `Aim2Score` and `Kick2Goal`, resulting in the task plan Petri net model depicted in Figure 9.

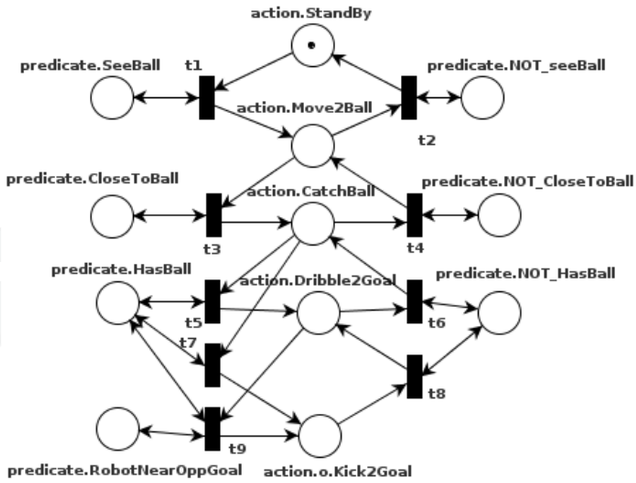



Fig. 9. Petri net model of the `Score_Goal` task.

Note that we use "`action.`" in the labels prefixes to denote action places. The label "`o.`" is used to denote which places should be marked in the desired final state of a given task model, denoting them as *output places*. Although this knowledge is not used yet, it will allow us to determine a task *desired-effects* in the future. There is no need to mark the places which are marked in the initial state, since this information is already given by the initial marking of the task model.

## 4. Analysis of Single-Robot Tasks

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 be both for logical (e.g. deadlocks) and probabilistic performance properties (e.g. probability of reaching a given state).

Furthermore, there are a number of properties that must be met during design time, which allow for some error detection at an early stage of development. As an example consider the boundedness of the net. Given that we are using predicate places, they 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 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 one, and only one, of predicate places $NOT\_p$ or $p$ must be an output place of transition $t$. If additionally one requires macro places to have at most one token, it results in a safe net requirement (i.e., have at most one token for all places, for all possible markings). Having the total number of tokens in the two places associated with a predicate equal to one is referred in Petri nets as a place invariant (Murata, 1989), which can also 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 also can also be extracted from the execution process in order to analyse the task a posteriori, and to further improve the models.

### 4.1 Expansion Process

The *Expansion Process* enables 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 these allow us to distinguish between the different types of places.
The expansion process is performed using Algorithm 4.1 while obeying the following set of rules:

- Predicate places with the same label are considered the same place;
- Macro places are always different places, regardless of their label;
- All transitions are different, regardless of their label.

The action macro places function as enabling places of all transitions on the associated models, i.e., if there is a token in the action macro place, then the transitions of the associated Petri net model are enabled (as long as the *running-conditions* and remaining input predicate places are true).

**Algorithm 4.1:** Full task Petri net model generation algorithm.

**Input**: Environment, task and action Petri net models
**Output**: Full Petri net model of the task

```
1  begin
2      Create an empty Petri Net, denoting it full-net;
3      foreach environment model do
4          Add the environment model to full-net;
5          Prefix all added transitions with the name of the model;
6      end
7      Add the task model to full-net;
8      foreach action macro place in full-net do
9          Add the Petri net model of the action associated with the action macro place to
           full-net;
10         Add an arc from the action macro place to all transitions in the added Petri net
           model;
11         Add an arc from all transitions in the added Petri net model to the action macro
           place;
12         Prefix the action macro place label with an "e" to denote that this is no longer a
           macro place, i.e., it has been expanded;
13         Prefix the labels of all added transitions with the name of the action;
14     end
15     Remove the tokens from all predicate places;
16 end
```

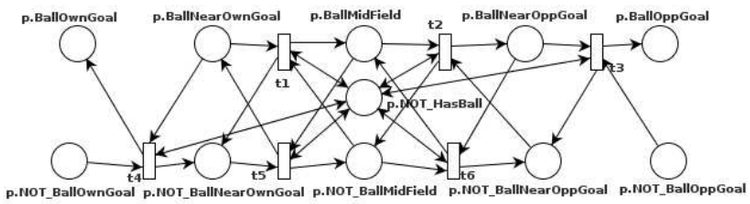
Prefixing the transitions with the model names during the expansion algorithm enables us to distinguish them while performing the analysis of the final model.
After having obtained the single Petri net, one needs to choose an initial state for the task by setting the number of tokens in the predicate places. Having set the initial marking of the net,

one can use available tools such as PIPE (Akharware, 2005) or TimeNET (Zimmermann, 2001) to study the task properties.

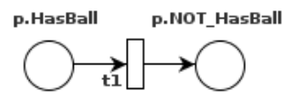
## 5. Execution of Single-Robot Tasks

In order to be able to execute the task plans developed within the framework, one needs to have a Petri net execution framework. In our case we have implemented such a framework in the decision layer of our MeRMaID middleware (Barbosa et al., 2007).
In MeRMaID, the sensorial part of the implementation keeps the predicates up to date (at least all the predicates that are revelant at any given state). Given a Petri net based task plan model, the *Petri net Executor* checks which transitions are enabled, considering the current selected actions and enabled predicates, and fires them accordingly. All actions that have tokens at any given moment are the actions that will be enabled. We have also taken advantage of part of the information provided at the Action Executor level, namely the *running-conditions*, so as to prevent running an action at the lower level when these are not satisfied.
The execution of the tasks can be monitored in order to assert and compare experimental results with the theoretical ones, allowing to check the models for errors or needed improvements.

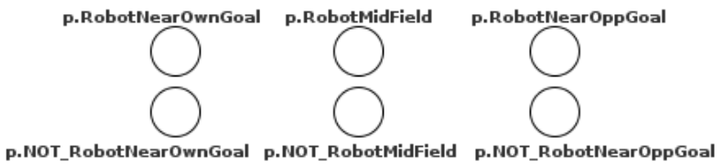
## 6. Single-robot Task Example



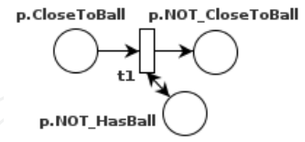

(a) Ball position model.




(b) `HasBall` model




(c) Robot Position model.




(d) `CloseToBall` model

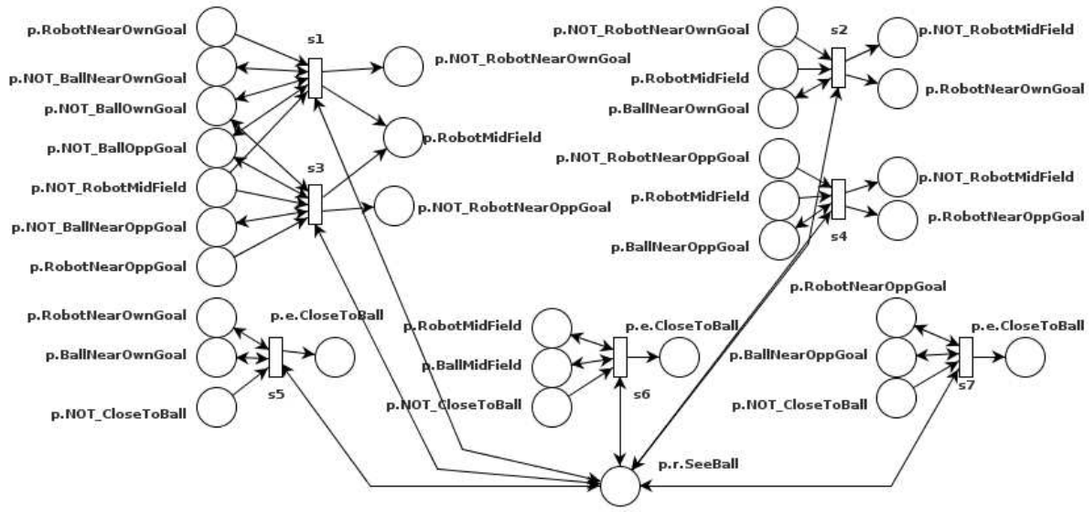
Fig. 10. Environment models for task `Score_Goal`.

To illustrate the framework application, we will detail a robotic soccer example using the single-robot task plan depicted in Figure 9. In this task we use the following predicates:

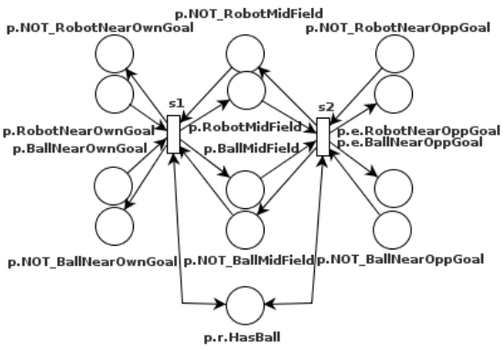
**Ball position:** `BallOwnGoal`, `BallNearOwnGoal`, `BallMidField`, `BallNearOppGoal`, `BallOppGoal`;

**Robot position:** `RobotNearOwnGoal`, `RobotMidField`, `RobotNearOppGoal`;

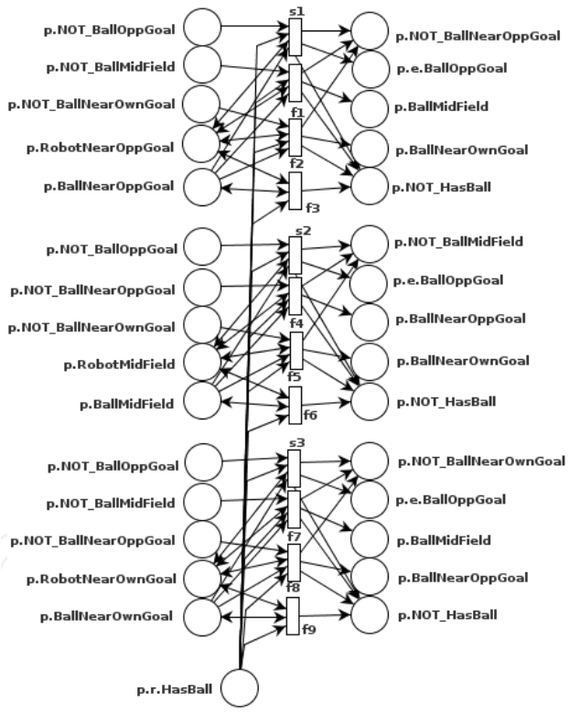
**Other:** `SeeBall`, `HasBall`, `CloseToBall`.




(a) `Move2Ball` model.




(b) `Dribble2Goal` model.




(c) `Kick2Goal` model.

Fig. 11. Action models

Predicate `HasBall` is true when the robot has posession of the ball, while `CloseToBall` is true when the robot is near the ball. The soccer field was divided in three regions, leading to the ball and robot position models, plus predicates `BallOwnGoal` and `BallOppGoal`, which are true when a goal is scored in our goal or in the opponent goal, respectively.

The environment models for this task are depicted in Figure 10. As can be seen from the models, we considered that the ball can be moved without being a direct result of the robot actions, or leave the proximity of the robot, as long as the robot does not hold the ball (Figure 10a and Figure 10d, respectively). The Petri net model in Figure 10b models the fact that the robot will eventually loose the ball posession. Furthermore, we considered that the robot could always see the ball, meaning predicate `SeeBall` is always true.
The actions used in this task are `StandBy`, `Move2Ball`, `CatchBall` (see Figure 8), `Dribble2Goal` and `Kick2Goal`, with the models being depicted in Figure 11. Note that we used labels $s_n$ for success transitions, and $f_n$ for failure transitions. The `StandBy` model is not shown because it is an empty model, i.e, since it does not perform changes in the environment, it does not contain any transition.
Table 1 gives a summary of the actions *running-conditions* and *desired-effects*. Recall that this information is available in the predicate labels of the action models, as explained in Section 3.2.

| Action | Running-conditions | Desired-effects |
|---|---|---|
| StandBy | - | - |
| Move2Ball | SeeBall | CloseToBall |
| CatchBall | SeeBall, Close2Ball | HasBall |
| Dribble2Goal | HasBall | RobotNearOppGoal, BallNearOppGoal |
| Kick2Goal | HasBall | BallOppGoal |

Table 1. Action properties.

The `Move2Ball` action is used by the robot to get near the ball. The model basically makes the robot position predicates change torwards the ball position predicate that is true, as long as the robot sees the ball. We include additional tests to avoid the robot moving to the ball when this is inside a goal.
The `CatchBall` action purpose is to grab the ball when the robot is close to the ball. As such, it makes the predicate `HasBall` become true, as long as the robot sees the ball and is near the ball.
The `Dribble2Goal` action is used by the robot to take the ball from its current position to near the opponent goal, thus changing the robot and ball position predicates until `BallNearOppGoal` and `RobotNearOppGoal` become true, as long as the robot has the ball.
Action `Kick2Goal` purpose is to score a goal, making the predicate `BallOppGoal` become true, as long as the robot has the ball. While actions `StandBy`, `Move2Ball`, `CatchBall` and `Dribble2Goal` do not explicitely include failures, the `Kick2Goal` action models does so. In action `Kick2Goal` we explicit modelled the fact that the robot can shoot torwards the goal from any place of the field, but the ball can end in any place of the field. Transitions $s_i$ correspond to success transitions, while transitions $f_i$ correspond to failures. By setting an higher rate to transition $s_1$ then $s_2$ and $s_3$, we are setting an higher probably of scoring when closer to the opponent goal. The other actions failures are modelled through the environment models. For instance, the predicate `HasBall` can become false at any time (see Figure 10b), leading to a failure of actions such as `CatchBall` and `Dribble2Goal`.
The rates used in the various models are as follows: 0.1 for the environment model rates except for the `HasBall` model, which we used 0.2; 1.0 for all action success transitions, except for transitions $s_2$ and $s_3$ in action `Kick2Goal`, where we used $1/4$ and $1/8$ respectively; for the failure transitions we used $1/4$. Note that since these are theoretical models, we consired

time to be measured in *time units*, meaning that an exponential transition with a rate of 1.0 will fire in average 1.0 times per *time unit* when enabled.

### 6.1 Results

We performed three transient tests of the task plan model shown in Figure 9 with TimeNET (Zimmermann, 2001), by considering different weights for transitions $t_5$ and $t_7$ (the only random switch available):

**Shoot_First:** by assigning weight 0 to transition $t_5$ and weight 1 to $t_7$, $t_5$ will never fire, meaning the robot goes from action `CatchBall` to action `Kick2Goal` without going through action `Dribble2Goal`, thus kicking to the goal as soon as it grabs the ball;

**Shoot_50_50:** by assigning weigh 1 to transitions $t_5$ and $t_7$, the robot chooses one of `Dribble2Goal` and `Kick2Goal` with probability 0.5, as soon as it grabs the ball while running action `CatchBall`;

**Shoot_Later:** by assigning weight 1 to transition $t_5$ and weight 0 to $t_7$, $t_7$ never fires, meaning the robot runs action `Kick2Goal` after having run action `Dribble2Goal` successfully. As such, the robot will only kick the ball when it has posession of the ball and it is near the opponent goal;

For each test we placed the robot near its goal and the ball in the field center, resulting in the following initial predicate state: `NOT_BallOwnGoal`, `NOT_BallNearOwnGoal`, `NOT_BallMidField`, `BallMidField`, `NOT_BallNearOppGoal`, `NOT_BallOppGoal`, `RobotNearOwnGoal`, `NOT_RobotMidField`, `NOT_RobotNearOppGoal`, `SeeBall`, `NOT_HasBall` and `NOT_CloseToBall`.

Since none of the actions performs changes on the environment when the ball is inside a goal, one can expect the task to include deadlocks, corresponding to scored goals. Qualitative analysis of the full task model confirmed that expectation, resulting in six deadlock states, corresponding to a goal scored from any of the three field regions into one of the two possible goals. Furthermore, we determined that the task is safe, having at most on token per place.

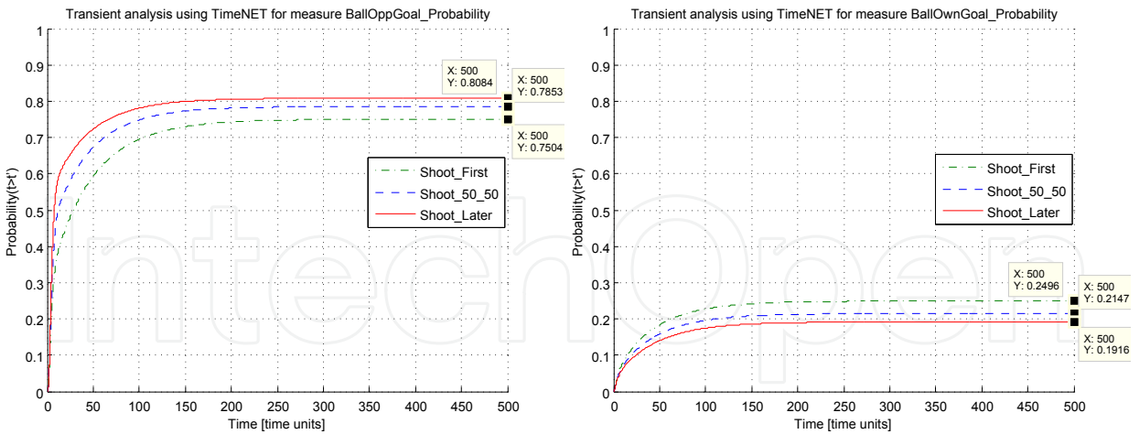
Each test consisted in analysing the task running from the initial marking until a deadlock occurred (goal scored), computing the number of expected tokens in places `BallOwnGoal` and `BallOppGoal` over time. This measure corresponds to the probability of having a goal scored in our goal or the opponent goal, yielding the results depicted in Figure 12.

The plots confirm that the ball must end in one of the goals, given that the sum of the probabilities of scoring in either goal when the system is already stationary is one.

As expected, kicking as soon as the robot grabs the ball leads to a lower scoring probability in the long term, since the the robot kicks from any position on the field, leading to more failures. However, analysing the initial time instants, depicted in Figure 13, shows that shooting the ball immediately leads to a higher scoring probability in the short term.

This is one example of interesting a priori results one can obtain using this framework. This knowledge can then be used in runtime, for instance, to change the weights of transitions $t_5$ and $t_7$ according to the score status and the game time left.

In qualitative terms, we determined that the task is safe, i.e., there is at most one or zero tokens in each place for all markings. Given that the action models are safe (see Definition 3.1), and the task `Score_Goal` is also safe (considering all possible predicate states), this results was expected. Furthermore, we also determined that all two places associated to a predicated formed place invariants wiht a total of 1 tokens, thus fully obeying Definition 2.5, as expected.




(a) Probability of scoring in the opponent goal. (b) Probability of scoring in our goal.

Fig. 12. Score goal probability evolution.

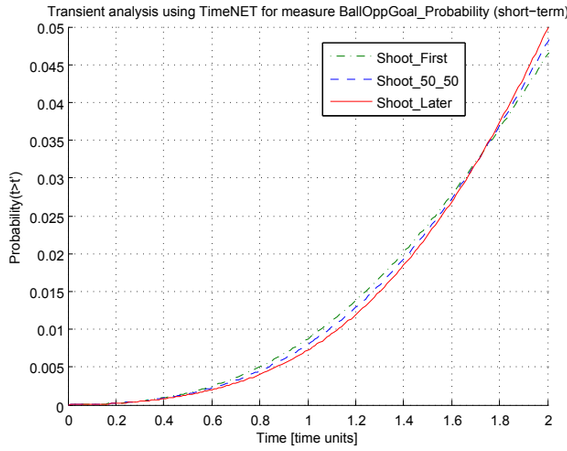



Fig. 13. Probability of scoring in the opponent goal (initial time instants).

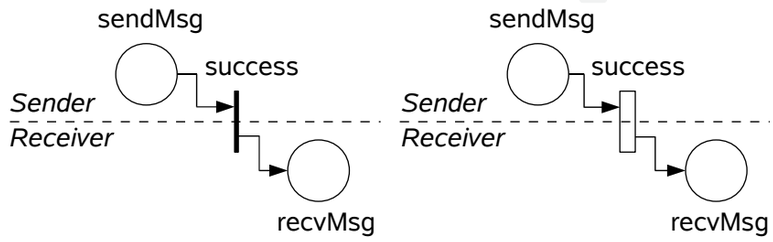
## 7. Modelling Multi-Robot Tasks using Petri Nets

The main difference between individual tasks and cooperative multi-robot tasks, is that some kind of synchronism must occur between the robots during task execution. This synchronism occurs through the use of communication, either explicitly or implicitly. Explicit communication happens when a robot (the sender) sends a message directly to the other robot(s), usually using Ethernet or wireless communications. Implicit communication happens when a robot, or robots, (the receivers) perceive some situation regarding the sender robot. As such, in order to model multi-robot tasks with our Petri Net based framework, we need first to introduce communication models.

### 7.1 Communication Models

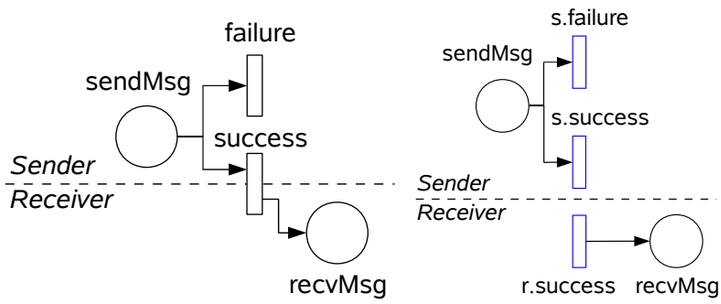
The major problem when using communication is the time information takes to go from the sender to the receiver, which, theoretically, can go from zero time to infinite time (communication failure). To model communication, we considered three different communication models,

which cover this time range. The base concept in these models is that a robot has a predicate place at a given value and wishes to transmit that information to a teammate. The teammate, upon receiving the information, gets its predicate updated to the same value as its teammate. The simplest communication model is presented in Figure 14a. Here the communication is considered instantaneous and always successful. Increasing the model complexity by adding a probabilistic arrival time for the communication, results in the model depicted in Figure 14b. In this case, communications are still considered always successful, but the amount of time it takes varies according to an exponential distribution. The full communication model is presented in Figure 14c. Here, we not only include a varying time delay, but also the possibility that the transition never reaches its destination, thus modelling communication failures.




(a) Deterministic communication model without failures.

(b) Communication model with exponentially distributed time and no failures.




(c) Full communication model with exponentially distributed time and failures.

(d) Separate view of the full communications model.

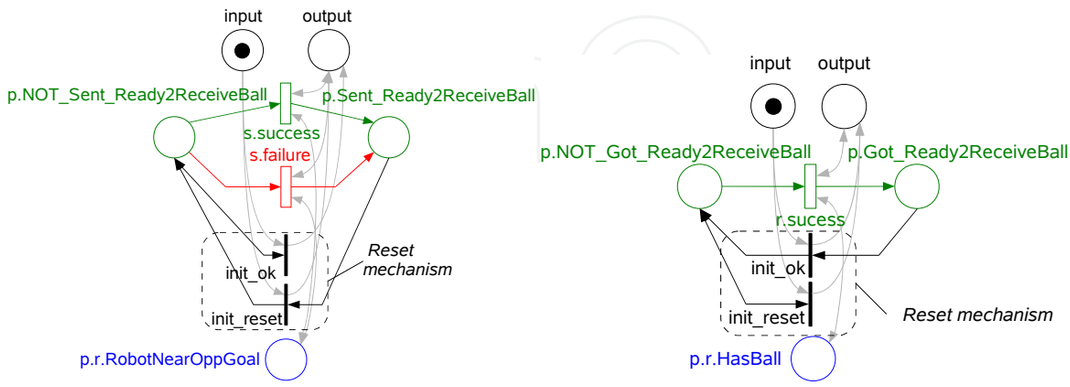
Fig. 14. Communication models.

Given the various communication models, we can choose which one to use, according to the context where the model is being applied and the properties we wish to analyse. When using the communication models, they will be seen in a distributed way to simplify the graphical view, as depicted in Figure 14d. Note that, when seen distributed, the communication transitions include a prefix to distinguish if the transition belongs to the sender or the receiver.

### 7.2 Communication Actions

In order to use the communication models to model direct communication between robots during a relational task, we define *Communication Actions*, which will be used to establish the required synchronisation. These actions, besides the specifications already defined for ordinary actions, include an additional reset mechanism. This mechanism is used to model

the fact that a communication event, when sent, is only received if the receiving robot, or robots, are expecting it, otherwise the event is ignored. For each sending communication model there will always be a receiving communication action model. As an example, see the Action Executor level models of actions `SendReady2Receive` and `RecvReady2Receive` in Figure 15a and Figure 15b respectively.




(a) Action `SendReady2Receive` model. (b) Action `RecvReady2Receive` model.

Fig. 15. Communication actions example.

Note that the *running-conditions* cannot be connected to the reset mechanism, as the token must be able to pass from place `input` to place `output` regardless of the current state.
The two depicted actions can be used to synchronise a two-robot behaviour, by running one in each robot.

### 7.3 Multi-Robot Task Plans

With the introduction of the communication models and communication actions, specifying a multi-robot task is similar to the specification of individual robot tasks. The major difference is that we need to use communication actions to ensure that the behaviours running during a multi-robot task execution are synchronised. For now we are assuming that the choice of running a relational task was already done, and focus on the multi-robot task execution analysis.

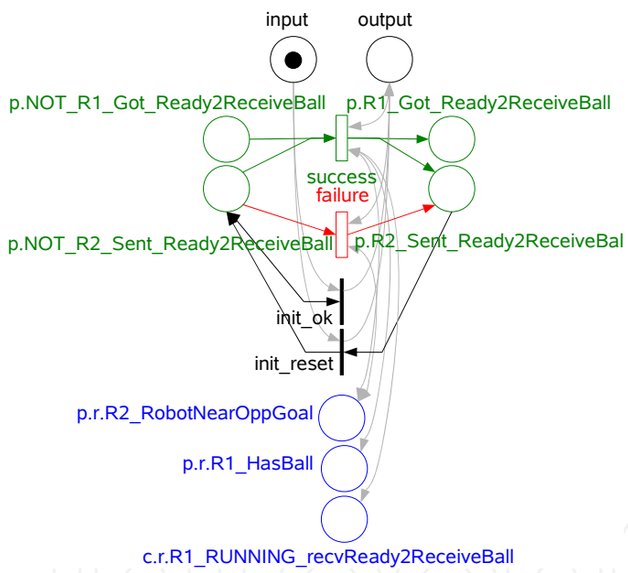
### 7.4 Analysis of Multi-Robot tasks

The analysis of multi-robot tasks in this framework is similar to the individual robot tasks case, adding the introduction of the communication models and actions. The difference relies on the fact that one needs to prefix the place labels identifying the robot they belong to, so as to distinguish between what is running in each robot, and the expansion of the communication actions need an additional step. Since each robot can run the same communication action at different times during the execution of a task plan, and the resulting Petri net used for analysis is static, simply expanding the communication actions would not work. This needed additional step corresponds to the creation of analysis versions of the communication actions, which is implemented through the following items:

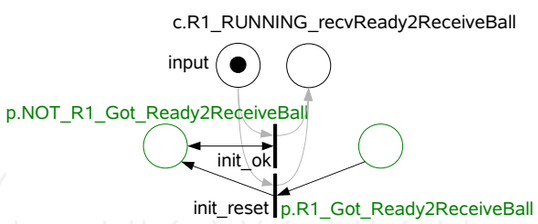
1. Move the transition associated with the communication from the receiving action model to the sending action model. The receiving transition is merged back with the successful sending transition, i.e, we obtain a single transition, located in the sending action, by connecting the arcs previously connected to the receiving transition to the successful sending transition;

2. Add the counter `RUNNING_commAction` to the receiving communication action with a token. Counter places have the prefix "c";
3. Make the added counter a *running-condition* of the sending communication action, connected only with the successful transition associated with the communication event.

The introduced counter will indicate the number of instances of receiving actions running in each robot and, most important, it will allow to track if a robot receiving action is running or not. This counter will be treated like a predicate at the expansion phase, i.e., every counter with the same label will be considered the same place.
With these analysis versions, the communicaton actions can be used anywhere in a robot task model, allowing for any receiving action to pair with the associated sending action, independently of where the communication actions appear in the task models. As an example, consider again the communication actions `SendReady2Receive` and `RecvReady2Receive` in a two robot setup, with their analysis versions depicted in Figure 16a and Figure 16b.
Naturally, it should be expected to have always at most one token in the counter, otherwise multiple actions were sending the same message simultaneously. This property can be computed during the analysis phase.




(a) Action `SendReady2Receive` model for analysis.




(b) Action `RecvReady2Receive` model for analysis.

Fig. 16. Communication actions example.

Although the action places are always considered different places, regardless of their label, these are also prefixed with the robot label, since different robots can have different action models. During the expansion of the macro places for analysis, all the communication action macro places are expanded into their analysis version instead of their original version. If we have more than two robots, then a selection mechanism must be used to select to which robot, or robots, the message is to be sent. The user never needs to see the analysis versions of the actions, since these are used only internally for analysis, and are automatically created from their original versions.

### 7.5 Multi-Robot Task Example

To illustrate the application of the framework to the multi-robot case, we will consider a pass example between two robots, the kicker and the receiver.

Given two tasks, `coordinatedKick`, for the kicker, and `coordinatedReceive`, for the receiver, a two-robot PASS task plan corresponds to a single `coordinatedPass` relational task, which consists of running both individual tasks in parallel, one in each robot. The key here is to make sure that both individual tasks run synchronously, either by implicit or explicit communication.

We assume that some higher level took the decision that the robots should commit with the coordinated pass, and will focus on the task execution analysis, keeping the critical sections synchronised.

For this example, we used the same list of predicates used in the single-robot example (see Section 6), plus predicates `Got_Ready2ReceiveBall` and `Sent_Ready2ReceiveBall`, associated to the communication actions. In terms of environment models we will use a ball position model (Figure 10a) and, per robot, one position model (Figure 10c), a lost ball model (Figure 10b) and a ball proximity model (Figure 10d).

For the PASS relational task plan we used actions `StandBy`, `Move2Ball` (Figure 11a) and `CatchBall` (Figure 8), used previously in the single robot example, plus the following actions:

**`Go2KickerPosture`:** The robot which has the ball, the kicker, moves to the kicker posture to be ready to pass the ball (Figure 17b), which is always considered to be near its own goal;

**`SendReady2Receive`:** The receiver acknowledges that it is ready to receive the ball (Figure 15a);

**`RecvReady2Receive`:** Waits for a communication from the receiver to know it is ready to receive the ball (Figure 15b);

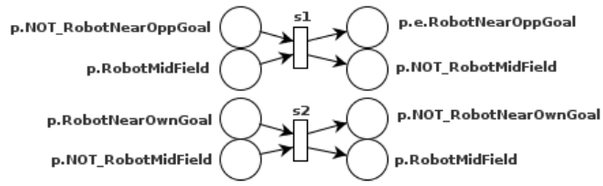
**`PassBall`:** Passes the ball to another robot. In this case we considered that passes are only done from near its own goal or the midfield to near the opponent goal (Figure 17c);

**`Go2ReceiverPosture`:** The robot moves to a destination posture, which is good for receiving the ball. We considered the receiving posture to be always near the opponent goal (Figure 17a).

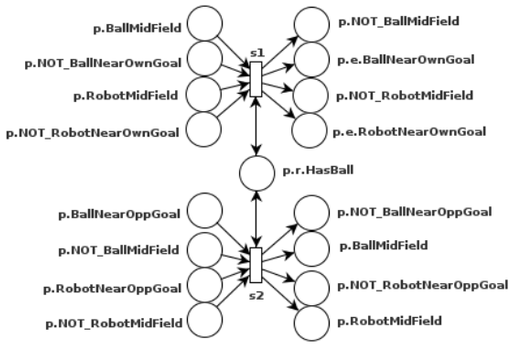
All transitions were removed from the action `Move2Ball` model except transition $s_7$, so as to allow the robot to be able to capture the ball only when near the opponent goal.

Task `CoordinatedKick` is obtained by running actions `Go2KickerPosture` and `RecvReady2Receive` in parallel, followed by action `PassBall` upon getting predicates `Ready2Pass` and `GotReady2Receive` to true. Task `CoordinatedReceive` is formed by a sequence of actions, starting with `Go2ReceiverPosture`, followed by `SendReady2Receive` when predicate `RobotNearOppGoal` gets true, ending with action `CatchBall` when `SentReady2Receive` gets true. Regarding communication, the relevant actions for the `coordinatedPass` relational task are `RecvReady2Receive` and `SendReady2Receive`, the two communication actions detailed previously. The Petri net models of both tasks are depicted in Figure 18a and Figure 18b.

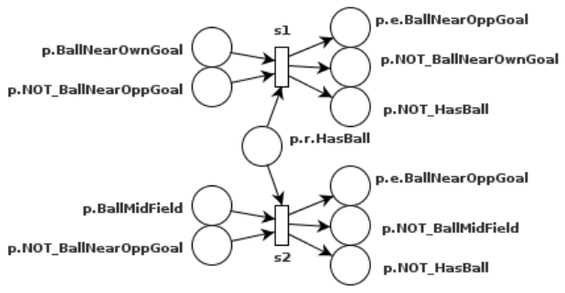
Given that our focus here is on the analysis of the execution of a multi-robot task, without using yet selection or commitment mechanisms, we consider a scenario where both robots are already set up for the execution of the pass. As such, the pass between the two robots can be obtained through the PASS task plan depicted in Figure 19.




(a) Action `Go2ReceiverPosture` model.

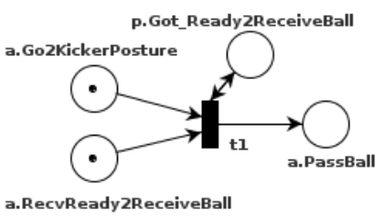



(b) Action `Go2KickerPosture` model.

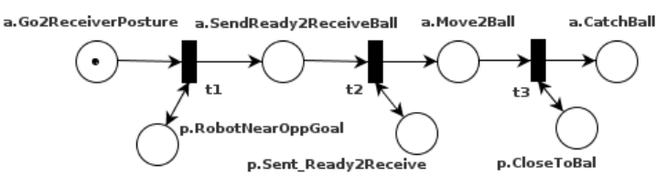



(c) Action `PassBall` model.

Fig. 17. Action models used in the multi-robot example.




(a) Task `CoordinatedKick` model.




(b) Task `CoordinatedReceive` model

Fig. 18. Task models used in the multi-robot example.

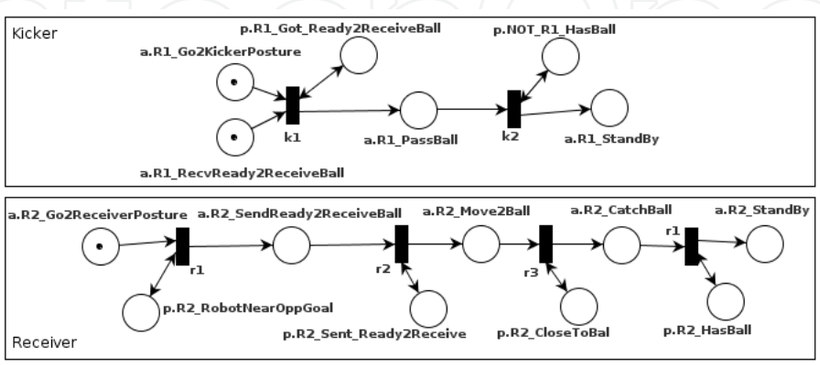



Fig. 19. PASS task plan.

### 7.6 Results

The setup used for the results consisted on placing both robots in the mid-field area, with robot R1 holding the ball, resulting in the following initial predicate state: `NOT_BallOwnGoal`, `NOT_BallNearOwnGoal`, `BallMidField`, `NOT_BallNearOppGoal`, `NOT_BallOppGoal`, `NOT_R1_RobotNearOwnGoal`, `R1_RobotMidField`, `NOT_R1_RobotNearOppGoal`, `R1_SeeBall`, `R1_HasBall`, `R1_CloseToBall`, `NOT_R1_Got_Ready2ReceiveBall`, `NOT_R2_RobotNearOwnGoal`, `R2_RobotMidField`, `NOT_R2_RobotNearOppGoal`, `R2_SeeBall`, `NOT_R2_HasBall`, `NOT_R2_CloseToBall`, and `NOT_R2_Sent_Ready2ReceiveBall`.

We analysed the PASS task plan success probability by monitoring the number of tokens in place `action.R2_StandBy`. Since robot R2 only reaches action `StandBy` if it was able to successfully receive the ball, reaching this action means the PASS task plan was successful.

The first results were conducted considering a deterministic environment (by removing the stochastic transitions from the environment models). Given that no failures were explicitely included in the action models, the only failure in this case is the communication failure. As such, the plan success probability should depend only on the relation between the communication failure and success rates, yielding:

$$P_{Plan\ success} = \frac{\lambda_{comm\ success}}{\lambda_{comm\ success} + \lambda_{comm\ failure}}$$

| Exp. # | Action success rates | Comm. success rates | Comm. failure rates | Plan success probability |
|---|---|---|---|---|
| **1** | 1 | 1 | 1 | 0.50 |
| **2** | 1 | 1 | 10 | 0.09 |
| **3** | 1 | 10 | 1 | 0.91 |
| **4** | 1 | 10 | 10 | 0.50 |
| **5** | 10 | 10 | 10 | 0.50 |

Table 2. Plan success probability vs transition rates with deterministic environment.

Table 2 shows the results obtained with different transitions rates for this setup, confirming the above statement. The graph showing the expected number of tokens in place `action.R2_StandBy` over time is shown in Figure 20 for experiments 1, 4 and 5. This graph shows that, although the stationary plan success probability only depends on the communication rates, increasing the success transition rates leads to a performance improvement in the short term.

Next we introduced additional failures by including the full models for `HasBall` and `CloseToBall` environment models for each robot, as shown in Figure 10b and Figure 10d. The ball position model was kept deterministic, without stochastic timed transitions. We tested this setup with different transition rates, obtaining the results show in Table 3.

In this case, increasing the communication success also increases the plan success probability as expected, but only to a certain point, as experiments 5 and 6 show. Only by increasing the remaining action transitions success rate can we further increase the plan success probability. In experiment 7, the success rates are much higher than the failure rates, leading to an almost 100% success probability.

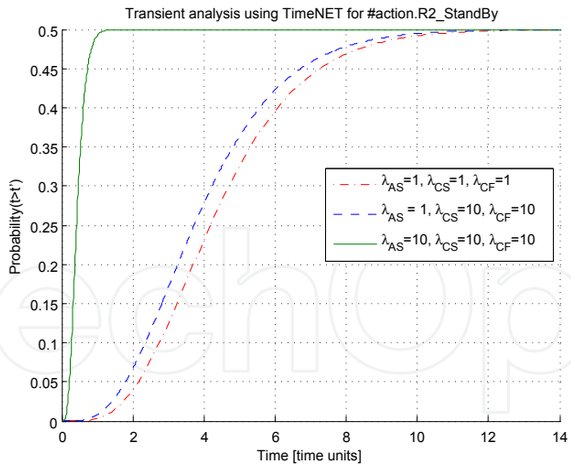



Fig. 20. PASS task plan success probability over time for different transition rates.

| Exp. # | Env. rates HasBall | Env. rates CloseToBall | Action success rates | Comm. success rates | Comm. failure rates | Plan success probability |
|---|---|---|---|---|---|---|
| **1** | 0.2 | 0.1 | 1 | 1 | 1 | 0.32 |
| **2** | 0.2 | 0.1 | 1 | 1 | 10 | 0.06 |
| **3** | 0.2 | 0.1 | 1 | 10 | 1 | 0.62 |
| **4** | 0.2 | 0.1 | 1 | 10 | 10 | 0.34 |
| **5** | 0.2 | 0.1 | 1 | 100 | 0.1 | 0.69 |
| **6** | 0.2 | 0.1 | 1 | 10000 | 0.0001 | 0.69 |
| **7** | 0.2 | 0.1 | 10 | 10000 | 0.0001 | 0.96 |

Table 3. Plan success probability vs transition rates with probabilistic environment.

Qualitatively we could determine, like in the single-robot example, that the task is safe, and that the predicate places form place invariants. Furthermore, as expected, both setups end always in deadlock, given that the tasks are sequential.

## 8. Conclusions and Future Directions

Petri nets provide a practical and intuitive way of modelling robotic tasks and associated components, being also appropriate to monitor the execution of tasks given their graphical nature. The fact that a GSPN is equivalent to a Markov chain brings an additional advantage by allowing the use of currently available tools and techniques to extract important a priori information about a given task.

Being able to model the actions more thoroughly at a lower level allows for mores realistic models, without compromising the analysis possibilities. Furthermore we can create all the models separately and build the task plan by creating a network of actions. This task plan can be ran directly on the robots for execution purposes and, for analysis purposes, we compose all the models that were designed separately onto one single Petri net, and analyse that net.

The introduction of communication models allowed the extension of the framework to multi-robot tasks, enabling a priori extraction of qualitative and quantitative properties of multi-robot tasks. Different communication models enable the study of the impact of a range of communication problems on the task success. We are currently improving the communication

action models to allow modelling broadcast type messages, which will allow for easier multi-robot tasks (with any number of robots) modelling.
Tests were performed using simulated robotic soccer scenarios which showed the applicability of the framework for both single-robot and multi-robot tasks. Qualitative properties, such as deadlock and safeness, and quantitative properties, such success probability over time, of the task were obtained from the full Petri net model.
In order to fully enable the use of the framework for multi-robot tasks, one still needs to implement selection and commitment mechanism. These mechanisms already exist in the literature (Cohen & Levesque, 1991; Palamara et al., 2009; van der Vecht & Lima, 2005) and we are working on incorporating them in our models. When analysing the complete models we will also be able to extract properties concerning the selection and commitment maintenance. We are currently implementing an identification algorithm which will allow building the action and environment models from real world data, leading to more realistic models. Furthermore, we plan to introduce observation models, allowing the use of the framework under scenarios without full observability.

## Acknowledgements

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

## 9. References

Akharware, N. (2005). *PIPE2: Platform Independent Petri Net Editor*, Master's thesis, Imperial College of Science, Technology and Medicine, University of London.

Barbosa, M., Ramos, N. & Lima, P. (2007). Mermaid - multiple-robot middleware for intelligent decision-making, *IAV2007 - 6th IFAC Symposium on Intelligent Autonomous Vehicles*.

Bernardinello, L. & Cindio, F. D. (1992). A survey of basic net models and modular net classes, *Advances in Petri Nets 1992, The DEMON Project*, Springer, pp. 304–351.

Cohen, P. R. & Levesque, H. J. (1991). Teamwork, *Noûs* **25**(4): 487–512.

Damas, B. D. & Lima, P. U. (2004). Stochastic Discrete Event Model of a Multi-Robot Team Playing an Adversarial Game, *Proceedings of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles*.

Dominguez-Brito, A. C., Andersson, M. & Christensen, H. I. (2000). A Software Architecture for Programming Robotic Systems based on the Discrete Event System Paradigm, *Technical Report CVAP244, ISRN KTH/NA/P–00/13–SE*, Centre for Autonomous Systems, KTH (Royal Institute of Technology).

Espiau, B., Kapellos, K., Jourdan, M. & Simon, D. (1995). On the Validation of Robotics Control Systems Part I: High Level Specification and Formal Verification, *Technical Report 2719*, INRIA.

Kosecka, J., Christensen, H. I. & Bajcsy, R. (1997). Experiments in Behaviour Composition, *Robotics and Autonomous Systems* **19**: 287–298.

Montano, L., García, F. J. & Villaroel, J. L. (2000). Using the Time Petri Net Formalism for Specification, Validation, and Code Generation in Robot-Control Applications, *The International Journal of Robotics Research* **19**(1): 59–76.

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

Palamara, P. F., Ziparo, V. A., Iocchi, L., Nardi, D. & Lima, P. (2009). Teamwork design based on petri net plans, pp. 200–211.

Petri, C. A. (1966). Kommunikation mit automaten, *Technical report*. English translation.

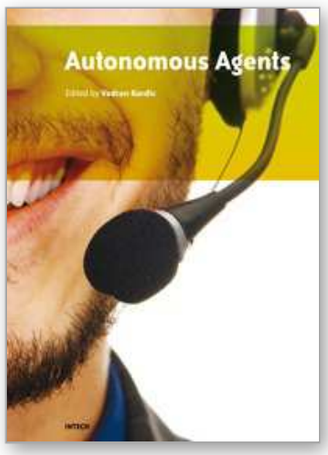
Röck, A. & Kresman, R. (2006). On Petri nets and predicate-transition nets, *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006*, pp. 903–909.

van der Vecht, B. & Lima, P. U. (2005). Formulation and Implementation of Relational Behaviours for Multi-robot Cooperative Systems, *Proceedings of RoboCup-2004: Robot Soccer World Cup VIII*, Springer-Verlag, pp. 516–523.

Viswanadham, N. & Narahari, Y. (1992). *Performance Modeling of Automated Manufacturing Systems*, Prentice Hall.

Zimmermann, A. (2001). TIMENET - a software tool for the performability evaluation with stochastic petri nets.

Ziparo, V. A. & Iocchi, L. (2006). Petri net plans, *Proceedings of the Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA'06)*, pp. 267–290.




**Autonomous Agents**
Edited by Vedran Kordic

ISBN 978-953-307-089-6
Hard cover, 130 pages
**Publisher** InTech
**Published online** 01, June, 2010
**Published in print edition** June, 2010

Multi agent systems involve a team of agents working together socially to accomplish a task. An agent can be social in many ways. One is when an agent helps others in solving complex problems. The field of multi agent systems investigates the process underlying distributed problem solving and designs some protocols and mechanisms involved in this process. This book presents a combination of different research issues which are pursued by researchers in the domain of multi agent systems.

**How to reference**
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Hugo Costelha and Pedro Lima (2010). Petri Net Robotic Task Plan Representation: Modelling, Analysis and Execution, Autonomous Agents, Vedran Kordic (Ed.), ISBN: 978-953-307-089-6, InTech, Available from: http://www.intechopen.com/books/autonomous-agents/petri-net-robotic-task-plan-representation-modelling-analysis-and-execution

INTECH
open science | open minds

**InTech Europe**
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

**InTech China**
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen