Designing Petri Net Supervisors from LTL Specifications

Bruno Lacerda* and Pedro U. Lima
Institute for Systems and Robotics – Instituto Superior Técnico
Lisbon, Portugal
Email: {blacerda,pal}@isr.ist.utl.pt

***Abstract*—We present a methodology to build a Petri net realization of a supervisor that, given a Petri net model of a (multi-)robot system and a linear temporal logic (LTL) specification, forces the system to fulfil the specification. The methodology includes composing the Petri net model with the Büchi automaton representing the LTL formula and trimming the result using a known method to reduce the size of the supervisor. Furthermore, we guarantee that the obtained supervisors are admissible by construction by restricting the LTL formulas that can be written to an appropriate subset. To illustrate the method, we provide an example on how to specify coordination rules for a team of simulated soccer robots.**

I. Introduction

The evolution of robotics in recent years, namely the development of more accurate sensors and actuators, is allowing the appearance of robot systems that perform increasingly complex tasks. While the first approaches to the deployment of such systems were specifically developed for the application at hand, the need to define formal approaches that are more general, in the sense that they provide a high-level language for the design of complex systems, while guaranteeing that certain safety, predictability, performance and robustness properties are satisfied, is becoming a central issue in the field.

In this work, we present a methodology that unites Petri nets (PN) [12] and linear temporal logic (LTL) [3], two formalisms particularly well suited for, respectively, the design of models of robot systems and the specification of rules for the systems to fulfil. This approach allows the designer to model the system as a PN that represents all possible behaviours of the robot(s) situated in its environment, and to specify a set of rules for the system to fulfil as LTL formulas. The PN model and the LTL formulas are then used to build a PN that restricts the behaviour of the system to one that is consistent with the specifications. Thus, the use of LTL can be seen as a bridge between the natural language specifications one has for the system, and the PN that is guaranteed to realize them. We argue that, for complex tasks, a direct construction of such PN is less intuitive, thus more susceptible to errors in the final result. This work is an extension of [11], which presented a similar method for finite state automata (FSA) models.

Due to its suitability to model concurrent systems and the wide range on analysis methods available, PNs have been used for the modelling and execution of (multi-)robot tasks [14, 2]. There has also been a considerable amount of work on the control of PNs. For example, in [7] a method where the specifications are written as linear constraints on the reachable markings of the system and the number of firings of each transition is defined and in [5] a study on the advantages and limitations of using PNs to realize supervisors is provided. We also refer the reader to [6], a survey on available methods for the control of PNs. In general, these works rely on disallowing the system to reach certain markings, with the specifications being written as linear inequalities. Writing these specifications, especially when they deal with coordination between different robots, can be quite cumbersome.

There have been several approaches to the use of temporal logic as a tool to specify and synthesize goal behaviours. The work presented in [13] introduces a planning algorithm over a domain given as an non-deterministic FSA where the states correspond to sets of propositional symbols and the goal is given as a temporal logic formula over those symbols. In [8], both the system and the goal specifications are encoded as a temporal logic formula, which is in turn translated into an FSA that satisfies it. In both of these works, contrary to our method, the temporal logic formulas are written only over the state space of the system, thus direct reasoning about sequences of events is not allowed. In [9], a motion planning method where the goals are defined as LTL formulas is presented. Using LTL to define the goals allows the specification of not only a goal region but also more intricate movements such as visiting a set of regions sequentially or travel between regions infinitely often. The work in [10] also deals with motion planning with temporal logic goals but allowing the robot to also react to sensor readings and perform actions other than moving. This approach also encodes both the system and the goal specifications as a temporal logic formula. In both of these works dealing with motion planning, a discretization of a linear system describing the robot motion capabilities is needed before the LTL specification can be enforced. In our discrete event system (DES) approach, particularly PNs, we do not take into account the continuous dynamics associated with the triggering of some of the events. This is reasonable when handling higher abstraction levels in robot task models, and significantly reduces the complexity of analysis. Moreover, we assume sensor-based primitive actions (e.g., moving while avoiding collisions), which handle/prevent some of the events

*Work partially funded by FCT (ISR/IST pluriannual funding) through the PIDDAC Program funds and FCT grant FRH/BD/45046/2008.

that might occur otherwise, and naturally discretize the set of actions that can be triggered by the PN. We claim that the use of PNs is more adequate to model the system than i) FSA, because the models are more expressive (PN languages are a superset of regular languages) and compact and ii) modelling the system directly in LTL, because the LTL formulas can become quite large for more complex systems, which has a big impact due to the complexity of solving SAT for LTL [3].

We start by providing the notions of logics and PN needed for our algorithm, in Sections II and III. Then, in Section IV, we define the composition between the Büchi automata obtained from the LTL specifications and the PN model of the system, and provide the rules to build the LTL formulas that guarantee admissibility. Finally, in Section V we show an application example to a simulated robot soccer scenario, followed by some conclusions and further work in Section VI.

II. Fundamental Logic Concepts

A. Propositional Logic

Propositional formulas are written over a set Π by applying the usual propositional connectives. In this work, we will consider that propositional formulas are in the disjunctive normal form (DNF), i.e., written as a disjunction of conjunctive clauses. A conjunctive clause is the conjunction of positive ($l = \pi,\ \pi \in \Pi$) and negative ($l = \neg\pi,\ \pi \in \Pi$) literals. For a given Π, we denote the set of all literals as $lit(\Pi)$, the set of all conjunctive clauses as $\mathscr{C}(\Pi)$ and the set of all DNF formulas as $dnf(\Pi)$.

Due to the fact that we will analyse the transitions of the PN directly, we will have to represent incomplete information about the state of the world after the firing of a given transition. To do that, we resort to *partial* valuations $v : \Pi \to \{0,1,\downarrow\}$. For a partial valuation, there may exist propositional symbols for which the truth value is not defined, i.e., $\pi \in \Pi$ such that $v(\pi) = \downarrow$. This means that v might not provide enough information to evaluate the truth value of the formula. We extend v to propositional formulas as follows:

- When the information provided by v is enough to guarantee that φ is satisfied, regardless of the assignment given to propositional symbols for which v is undefined, $[\![\varphi]\!]_v = true$;
- When the information provided by v is enough to guarantee that φ cannot be satisfied, regardless of the assignment given to propositional symbols for which v is undefined, $[\![\varphi]\!]_v = false$;
- When the truth value of φ cannot be evaluated with only the information provided by v, $[\![\varphi]\!]_v$ denotes the formula composed of the conjunctive clauses that contain the literals for which the truth value is undefined.

Consider the formula $\varphi = (\neg p \wedge r) \vee (p \wedge \neg q \wedge \neg r) \in dnf(\{p,q,r\})$ and the valuation v such that $v(p) = 1$, $v(q) = 0$ and $v(r) = \downarrow$. We have that $[\![\neg p \wedge r]\!]_v = false$, because $v(p) = 1$, and $[\![p \wedge \neg q \wedge \neg r]\!]_v = \neg r$ because v does not provide information about r. Hence, we also have $[\![\varphi]\!]_v = \neg r$. Note that, for a conjunctive clause, it suffices that v does not satisfy one of its literals, regardless of it not satisfying or being inconclusive for the others, for v not to satisfy that conjunctive clause and, for a DNF formula, it suffices that v satisfies a conjunctive clause for v to satisfy that formula.

We will also need the notion of partial valuation generated by a conjunctive clause φ, which is a function $v_\varphi : \Pi \to \{0,1,\downarrow\}$ that assigns 1 to the positive literals of φ, 0 to the negative literals of φ and $\downarrow$ to the remaining propositional symbols in Π. For example, formula $\varphi = (\neg p \wedge r) \in \mathscr{C}(\{p,q,r\})$generates the valuation v_φ such that $v_\varphi(p) = 0$, $v_\varphi(q) = \downarrow$ and $v_\varphi(r) = 1$.

B. Linear Temporal Logic

LTL formulas are written over a set Π using the propositional connectives, as in propositional logic, plus a set of temporal connectives. This set is composed of the next (X), eventually (F), always (G) and until (U) connectives. LTL formulas are evaluated over ω-strings of sets of propositional symbols $\sigma = \sigma_0\sigma_1\sigma_2... \in (2^\Pi)^\omega$. For a given state σ_i, $X\varphi$ is satisfied if φ is satisfied in the next state σ_{i+1}, $F\varphi$ is satisfied is there exists a state σ_j with $j \geq i$ that satisfies φ, $G\varphi$ is satisfied if all states σ_j with $j \geq i$ satisfy φ and $\varphi U \psi$ is satisfied if there exists $j \geq i$ such that σ_j satisfies ψ and σ_k satisfies φ for all $i \leq k < j$. If state σ_0 satisfies a formula φ, we say that the sequence σ satisfies φ, denoted $\sigma \Vdash \varphi$.

A very useful property of LTL is that, for any formula φ, there exists a (non-deterministic) Büchi automaton (BA) B_φ accepting exactly the ω-strings that satisfy φ. We will translate LTL formulas to BA using one of the most efficient translation algorithms, LTL2BA, described in [4]. The automaton obtained by this implementation is a tuple $\langle Q, 2^\Pi, f, q_0, Q_f \rangle$ where Q is the set of states, 2^Π is the alphabet, f is the non-deterministic transition function, q_0 is the initial state and Q_f is the set of final states. An ω-string is accepted by the BA if it generates a run that goes through at least one accepting state infinite times. DNF formulas are used to describe the transition labels in a more compact way, for example, if $e \wedge \neg d$ is a transition label, then all elements of 2^Π that contain e and do not contain d are allowed in that transition. Also, the resulting automaton is *trimmed*, i.e., all its states can be reached and there is a path between each state and at least one accepting state. This means that the finite behaviour of the observer[1] of a BA obtained from φ using the LTL2BA algorithm is consistent with φ, in the sense that any string generated by a run of the observer is a prefix of an ω-string that satisfies φ.

III. Petri Nets

A. General Definitions

We will model our system as a Petri net (PN) with event labels associated with the transitions and a subset of the places representing the truth value of symbols that are used to describe the state of the system. The alphabet for our temporal logic specifications will be the union of the event set and these state description symbols. We start by introducing the

[1]The observer of a non-deterministic automaton G is its deterministic version, built using the known power-set construction [1].

PN basics and then show how the state description symbols are added to the model and how one can obtain a description of the state for a marking and for the firing of a transition.

A PN structure is a directed bipartite graph represented by a tuple $G = \langle P, T, W^+, W^-, M_0 \rangle$. The two types of nodes are given by sets P (places) and T (transitions). Matrices $W^-, W^+ : T \times P \to \mathbb{N}$ represent, respectively, the arc weights from places to transitions, and the arc weights from transitions to places. The vector $M_0 : P \to \mathbb{N}$ is the initial marking. A marking is a distribution of *tokens* among places and represents a state of the system.

Given a transition t, we define the vectors ${}^\bullet t : P \to \mathbb{N}$ (preset of t) and $t^\bullet : P \to \mathbb{N}$ (postset of t) as ${}^\bullet t(p) = W^-(t,p)$ and $t^\bullet(p) = W^+(t,p)$. In a given marking M, t is active if ${}^\bullet t \leq M$. If t is active in M, it can fire, evolving the PN to marking $M' = M - {}^\bullet t + t^\bullet$, i.e., when t fires it consumes the tokens in its preset and adds tokens to its postset, according to the respective weights. We denote this as $M \xrightarrow{t} M'$. The set of reachable markings in G, starting in M_0 and following the firing rule above, is denoted $R(G)$.

We will be interested in the languages generated by PNs, hence we add labels to the transitions. This is done by adding to the PN structure an event set E and a labelling function $\ell : T \to E$, that assigns to each transition a label from E. The language generated by a labelled PN G is defined as:

$$\mathscr{L}(G) = \Big\{ \ell(t_1)...\ell(t_n) \in (2^E)^* \mid \text{exists } M_1,...,M_n \text{ such that } M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} ... \xrightarrow{t_n} M_n \Big\}$$

As we will see later, the elements of this language will be the input for our PN supervisors.

B. Petri Net Model of a Robotic Task

For the system models, we require more information than only the sequence of events. We want to also have some kind of description of the states that are visited while executing those events. This is done by partitioning the places into a set P_D that represents the truth value of state description symbols and a set P_g of general places that do not influence the state propositional description, and adding the following to the labelled PN tuple:

- A set D of state description symbols;
- A bijection[2] $\mu : lit(D) \to P_D$, where $\mu(l)$ is a place that has one token whenever l is satisfied;
- A relation $K \in \mathscr{C}(E \cup D) \times \mathscr{C}(D)$ between conjunctive clauses of $E \cup D$ and conjunctive clauses of D. Intuitively, $(\varphi, \psi) \in K$ means that whenever φ is satisfied in our model, then ψ is also satisfied.

The set of propositional symbols associated with a PN system model is $\Pi = E \cup D$. We assume that all our PN system models are propositionally consistent, i.e., for all $M \in R(G)$ and $d \in D$, $M(\mu(\neg d)) + M(\mu(d)) = 1$. Propositionally consistent PN system models always have, for each d, one token in one of the places representing a truth value for d and zero tokens in the other. Also, we add a knowledge base K, which states relations between different state description symbols that the designer knows will always hold in the PN. We can now define the set of true state description symbols and the corresponding valuation describing the truth value of all elements of D in a given marking M as $D_M = \{d \in D \mid M(\mu(d)) = 1\}$ and, for $d \in D$, $v_M(d) = M(\mu(d))$, respectively. Note that v_M is a valuation which is defined for all state descriptions symbols $d \in D$ and undefined for all events $e \in E$.

We are now in conditions to define the language generated by a PN system model:

$$\mathscr{L}^D(G) = \Big\{ (\{\ell(t_1)\} \cup D_{M_1}) ... (\{\ell(t_n)\} \cup D_{M_n}) \in (2^{E \cup D})^* \mid M_0 \xrightarrow{t_1} ... \xrightarrow{t_n} M_n \Big\}$$

The generated language is a sequence where each element is of the form $\{e, d_1, ...d_n\}$, where $e \in E$ is the event that occurred and $d_1, ..., d_n \in D$ are the state description symbols that are true in the marking reached after the occurrence of that event. Note that $\mathscr{L}^D(G)$ does not include the initial marking M_0. To deal with this, we add an *init* place and an *init* transition to all our PN system models and change their initial marking. The *init* place has initial marking equal to 1, while all other places have initial marking equal to 0. The system always starts by firing the *init* transition, which consumes the token of the *init* place and distributes tokens to the other places of the PN, according to M_0.

Since we want to avoid building the whole state space of the PN to apply our method, we will analyse the PN structure directly. To perform this analysis, in addition to the valuations v_M generated by markings $M \in R(G)$, we will also be interested in the valuations $v_t : E \cup D \to \{0, 1, \downarrow\}$, obtained by the firing of transition $t \in T$. For events $e \in E$, v_t is always defined, being 1 for event $\ell(t)$ and 0 for all others. For $d \in D$, $v_t(d) = 1$ if the place representing that d is true is in the postset of t (i.e., $t^\bullet(\mu(d)) = 1$), $v_t(d) = 0$ if the place representing that d is false is the postset of t and $v_t(d) = \downarrow$ otherwise. Note that, for all state description symbols d and markings M that can be reached immediately after firing t, if $v_t(d)$ is defined then $v_M(d) = v_t(d)$. Hence, v_t represents the information that is guaranteed to hold immediately after the firing of t, regardless of the marking from which t was fired.

One can augment this valuation, by defining truth values to additional state description symbols, according to K:

$$v_t^K(\pi) = \begin{cases} v_t(\pi) & \text{if } v_t(\pi) \neq \downarrow \\ v_\psi(\pi) & \text{if there exists } (\varphi, \psi) \in K \text{ such that } v_t \Vdash \varphi \text{ and } v_\psi(\pi) \neq \downarrow \\ \downarrow & \text{otherwise} \end{cases}$$

To ensure that the above function is well-defined, we assume that the information given by K is *consistent* with all the reachable markings of the PN, i.e., for all reachable markings M, M' and transitions t such that $M \xrightarrow{t} M'$ and for all $(\varphi, \psi) \in K$:

$$\text{If } v_t \Vdash \varphi \text{ then } v_{M'} \Vdash \psi$$

[2]Hence, each $p \in P_D$ corresponds to exactly one literal $l \in lit(D)$, thus $|P_D| = 2|D|$, where $|.|$ denotes the number of elements of the set.

The above statement means that, whenever we fire a transition t satisfying φ, the PN evolves to a marking that satisfies ψ. One should note that it is up to the designer to build a relation K that is consistent. This relation is not required for the application of our method, but it allows us to augment the valuation associated with a transition, which, as we will see later, may reduce the number of transitions in the PN realizing the supervisor.

IV. Supervisor Synthesis

A. Supervisory Control concepts

The purpose of supervisory control (SC) is to restrict the open-loop uncontrolled behaviour of a system – in our case modelled as a PN G – to an admissible language $\mathscr{L}_a \subseteq \mathscr{L}(G)$. We start by partitioning the event set E into set E_c of controllable events – the events that the supervisor can disable – and set E_{uc} of uncontrollable events – events that are not of the "responsibility" of the robot itself, e.g., changes in the environment which are not related to the robot's actions.

Formally, a supervisor is a function $S : \mathscr{L}(G) \to 2^E$ that, given $s \in \mathscr{L}(G)$, outputs the set of events G can execute next (*enabled events*). We only allow admissible supervisors, that is, supervisors that never disable uncontrollable events which are active for the uncontrolled system. In this work, we will use *modular* SC, where the modular supervisor S is represented by n supervisors $S_1, ..., S_n$ and the set of enabled events for S is given by the intersection of the enabled events for each supervisor S_i. Each S_i will be realized as a deterministic PN[3] which runs in parallel with the system, executing the same events, and outputting the set of enabled events given by the labels of its current active transitions. The fact that we are using PNs to realize the supervisor function requires them to be deterministic: if a PN is not deterministic, then the value of $S(s)$ is not uniquely defined, as it depends on the non-deterministic choices made when firing the transitions corresponding to the events. Hence, we require the PN models of the system to be deterministic and we build the observer automaton of the non-deterministic BA *on-the-fly*, during the composition. This guarantees that the obtained supervisor is also deterministic. Given a PN system model G and PN realizations of n supervisors $S_1, ..., S_n$, we denote the language of events plus state description symbols generated by G when modularly supervised by $S_1, ..., S_n$ as $\mathscr{L}^D(G/S_1, ...S_n)$.

B. Constructing the LTL-based PN supervisor

Problem 1: Given a PN system model $G = \langle P, T, W^+, W^-, M_0, E, \ell, D, \mu, K \rangle$ and a set of LTL formulas $\Phi = \{\varphi_1, ...\varphi_n\}$ written over the set $\Pi = E \cup D$, build n PN supervisors $S_{\varphi_1}, ..., S_{\varphi_n}$ such that the generated language of G when modularly supervised by $S_{\varphi_1}, ..., S_{\varphi_n}$ is the largest language contained in $\mathscr{L}^D(G)$ such that:

$$\text{If } s \in \mathscr{L}^D(G/S_{\varphi_1}, ..., S_{\varphi_n}) \text{ then there exists}$$
$$\sigma_1,\ ...,\ \sigma_n \in (2^{E \cup D})^\omega \text{ such that } \begin{cases} s\sigma_1 \Vdash \varphi_1 \\ \vdots \\ s\sigma_n \Vdash \varphi_n \end{cases}$$

The ω-string $s\sigma$ is the concatenation of string s with ω-string σ. To solve this problem, we will define a composition function that given the PN model of the system G and the (non-deterministic) BA B_φ, builds a PN system model S_φ such that if $s \in \mathscr{L}^D(S_\varphi)$ then there exists $\sigma \in (2^{E \cup D})^\omega$ such that $s\sigma \Vdash \varphi$. Thus, for a set of formulas $\Phi = \{\varphi_1, ..., \varphi_n\}$, by running G modularly supervised by $S_{\varphi_1}, ..., S_{\varphi_n}$, we solve Problem 1. The construction of this PN follows Algorithm 1.

The algorithm creates a PN that simulates a run in parallel of the PN model of the system and the observer of the BA, where a transition t can only fire in parallel with a transition of the BA observer labelled by ψ when we are ensured that the marking to which the PN evolves satisfies ψ. Thus, we only allow the firing of transitions that lead to sequences of events plus state descriptions that are in conformity with the BA transitions (hence are consistent with the LTL formula).

We start by analysing the initial state $\{q_0\}$ of the observer, and only analyse states $Q'' \in 2^Q$ that are attained during the execution of the algorithm (lines 22–26). When analysing a state Q'' of the observer, we start by building, for each state q of the BA, the DNF formula ℓ_q (lines 7–9), which represents all the members of $2^{E \cup D}$ that can take us from a state in Q'' to q. This DNF formula is simply the disjunction of the DNF formulas labelling transitions from elements of Q'' to q. Then, for each $t \in T$, we apply v_t^K to each ℓ_q (lines 12–18). In the cases where v_t^K satisfies ℓ_q, we are guaranteed that after the firing of t, the observer will go from Q'' to a state that contains q. Hence, we add q to set *next_guaranteed_states*. In the cases where v_t^K is inconclusive for ℓ_q, we know that the observer might be able to go from Q'' to a state that contains q in certain situations, depending on the marking from which t was fired (formula $[\![\ell_q]\!]_{v_t^K}$ encodes the markings for which the observer will go to a state that contains q). Hence, we add q to set *next_possible_states*.

After the creation of these sets, if both of them are empty, it means that t can never occur when the observer is in state Q'', i.e., when there is a token in the place of the PN supervisor representing state Q''. If at least one of them is not empty, it means that there are situations where t can fire when there is a token in the place representing that the observer is in state Q''. The observer will always evolve to a superset of *next_guaranteed_states*, because whenever t fires, there is a DNF transition label from a state in Q'' to a state in *next_guaranteed_states* that is satisfied. Then, we check for each $Q' \in 2^{next\_possible\_states}$, what are the markings from which we jump to Q' when t is fired (lines 20–36). These markings are encoded by the DNF formula ψ (line 27), and satisfy all the labels that take us to a state in Q' and do not satisfy any of the labels that take us to a state in $next\_possible\_states \setminus Q'$.

[3] A PN G is deterministic if in all of its reachable markings $M \in R(G)$, if $M \xrightarrow{t} M'$, $M \xrightarrow{t'} M''$ and $M' \neq M''$, then $\ell(t) \neq \ell(t')$.

Algorithm 1 Büchi/System composition

Input: PN $G = \langle P, T, W^+, W^-, M_0, E, \ell, D, \mu, K \rangle$ and LTL formula φ
Output: PN $S_\varphi = \langle P', T', W^{+'}, W^{-'}, M'_0, E, \ell', D, \mu, K \rangle$

1: $B_\varphi = \langle Q, 2^{E \cup D}, f, q_0, Q_f \rangle \leftarrow LTL2BA(\varphi)$
2: $P' \leftarrow P$; $M'_0 \leftarrow M_0$
3: $states\_queue.push(\{q_0\})$
4: add place p with label $\{q_0\}$ to P'; $M'_0(p) \leftarrow 1$
5: **while** $states\_queue \neq \emptyset$ **do**
6: $current\_states \leftarrow states\_queue.pop()$
7: **for all** $q \in Q$ **do**
8: $\ell_q \leftarrow \bigvee_{q' \in current\_states} \ell(q', q)$ $\{\ell(q', q)$ is the label ψ of transition $f(q', \psi) = q$, if it is defined or $false$ otherwise$\}$
9: **end for**
10: **for all** $t \in T$ **do**
11: $next\_guaranteed\_states \leftarrow \emptyset$; $next\_possible\_states \leftarrow \emptyset$
12: **for all** $q \in Q$ **do**
13: **if** $[\![\ell_q]\!]_{v_t^K} = true$ **then**
14: add q to $next\_guaranteed\_states$
15: **else if** $[\![\ell_q]\!]_{v_t^K} \neq false$ **then**
16: add q to $next\_possible\_states$
17: **end if**
18: **end for**
19: **if** $next\_guaranteed\_states \cup next\_possible\_states \neq \emptyset$ **then**
20: **for all** $Q' \in 2^{next\_possible\_states}$ **do**
21: $next\_possible\_label \leftarrow next\_guaranteed\_states \cup Q'$
22: **if** $\nexists p \in P'$ with label $next\_possible\_label$ **then**
23: $states\_queue.push(next\_possible\_label)$
24: add place p' with label $next\_possible\_label$ to P'
25: $M'_0(p') \leftarrow 0$
26: **end if**
27: $\psi \leftarrow (\bigwedge_{q' \in Q'} [\![\ell'_q]\!]^K_{v_t}) \wedge (\bigwedge_{q' \in next\_possible\_states \setminus Q'} \neg [\![\ell'_q]\!]^K_{v_t})$
28: **if** $\psi \neq false$ **then**
29: **for all** conjunctive clauses $\gamma \in DNF(\psi)$ **do**
30: add transition t' to T'; $\ell'(t') \leftarrow \ell(t)$
31: $read\_places \leftarrow \{\mu(l) \in P'_D \mid l$ occurs in $\gamma)\}$
32:

$$ {}^\bullet t'(p) \leftarrow \begin{cases} {}^\bullet t(p) & \text{if } {}^\bullet t(p) > 0 \\ 1 & \text{if } p \text{ has label } current\_states \text{ or } p \in read\_places \\ 0 & \text{otherwise} \end{cases} $$

33:

$$ t'^\bullet(p) \leftarrow \begin{cases} t^\bullet & \text{if } t^\bullet(p) > 0 \\ 1 & \text{if } p \text{ has label } next\_possible\_label \text{ or } p \in read\_places \\ 0 & \text{otherwise} \end{cases} $$

34: **end for**
35: **end if**
36: **end for**
37: **end if**
38: **end for**
39: **end while**

Hence, whenever we fire t and already are in a marking that is consistent with ψ (i.e., a marking M such that v_M satisfies the part of ψ for which v_t is undefined), the observer goes from state Q'' to state $next\_guaranteed\_states \cup Q'$. This means that whenever t fires in such a marking, a token must be taken from the place representing observer state Q'' and placed in the place representing observer state $next\_guaranteed\_states \cup Q'$. Hence, for each conjunctive clause γ in ψ, we create a transition (t, γ)[4] in the supervisor, with the following 3 types of arcs:

1) *System arcs*, i.e., the same arcs as in t. These arcs take into account the evolution of the system when t is fired;
2) *Büchi arcs*, i.e., arcs that consume a token from the place representing the observer state Q'' and put one token in the place representing the observer state $next\_guaranteed\_states \cup Q'$. These arcs take into account the evolution of the observer of the BA;
3) *Reflexive-arcs*, i.e., arcs going to and from the places in the set $read\_places$ (line 31) obtained from γ. This set contains the places that must have a token in order to guarantee that γ is satisfied after the firing of t. These arcs guarantee that (t, γ) only fires when we are in a marking consistent with γ (which implies that we are in a marking that satisfies ψ since it is in the DNF).

C. Deleting Dead Transitions

To take advantage of the distributed state representation of PNs, we build the supervisor by analysing the PN structure directly. Specifically, we analyse each transition of the PN against all the transitions in the output of each analysed state of the observer of the BA. This can cause the creation of transitions in the PN realization of the supervisor that are never active, i.e., *dead* transitions. In order to reduce the size of the supervisors, we trim these transitions, by using a known algebraic method [1]: If the following integer linear program (ILP) has no solution, we are guaranteed that t is a dead transition and we can delete it. We note that this method is not complete, in the sense that there might be some dead transitions that are not deleted by applying it.

$$ \begin{array}{ll} \text{find} & w \\ \text{subject to} & M'_0 + (W^{+'} - W^{-'})w \geq {}^\bullet t \\ & w \in \mathbb{N}^{|T|} \end{array} $$

Given the high complexity of ILPs and the need to solve one for each transition in the supervisor, we relax the problem to a linear program (LP), with real variables. For each transition t, if the LP has no solution, then the ILP does not have one either and we delete t. Using this relaxation, transitions for which there is a solution for the LP but there is none for the ILP are not deleted. In spite of that, in our experiences, several instances of the ILP were not solved in a reasonable amount of time, and the LP relaxation always gave the same results as an ILP with a bound on runtime (where transitions for which a solution is not found until the bound are not deleted).

D. Building an Admissible Supervisor

A key limitation of the use of PNs to realize supervisors is related to the notion of supervisor admissibility. Namely, they are not closed under the extraction of the *supremal*

[4]Each γ represents a different situation for which t can fire.

controllable sublanguage [5]. Intuitively, this means that if our PN supervisor is not admissible, we do not have a way to effectively restrict its behaviour so that it becomes admissible.

Given this fact, we will restrict the LTL formulas used to specify behaviours such that the supervisor admissibility is guaranteed by construction. Given a PN system model G, we define the set lit_c of *controllable state description literals* as:

$$lit_c = \{l \in lit(D) \mid \text{for all } t \in T_{uc}, \text{ if } {}^\bullet t(\mu(l)) = 1 \text{ then } t^\bullet(\mu(l)) = 1\}$$

The set T_{uc} is simply the set of transitions with an uncontrollable event labels, i.e., $T_{uc} = \{t \in T \mid \ell(t) \in E_{uc}\}$. The set lit_c contains the literals associated to places for which the firing of uncontrollable transitions keeps their marking intact (note that this includes the places that are not in the preset of any uncontrollable transition). This means that our supervisor can always enforce that the truth value of elements l of lit_c cannot be altered while being admissible. This is due to the fact that for all $l \in lit_c$, all the transitions that have place $\mu(l)$ in its preset (and not in its postset) correspond to controllable events. The "admissible" LTL specifications φ must be of one of the following types:

1) $G\psi$, where ψ is a propositional formula in the DNF where only literals in lit_c, or literals $\neg e$ with $e \in E_c$ can occur. These formulas specify propositional logic relations between state descriptions and controllable events that should always be satisfied in all runs of the supervised PN. For example, we might want that two given robots do not move at the same time or that a certain controllable event never occurs;
2) $G(\gamma \Rightarrow X\psi)$, where γ is any propositional formula in the DNF and ψ is a propositional formula in the DNF where only literals in lit_c that also appear in γ, or literals $\neg e$ with $e \in E_c$ can occur. These formulas specify that the propositional logic formula ψ must be satisfied exactly after condition γ is met. For example, we may specify that if a robot is moving forward and there is no obstacle in front of him, it should continue moving forward;
3) $G(\gamma \Rightarrow X(\psi U \gamma'))$, where γ and γ' are any propositional formulas in the DNF and ψ is a propositional formula in the DNF where only literals in lit_c that also appear in γ, or literals $\neg e$ with $e \in E_c$ can occur. These formulas specify that after condition γ is satisfied, the propositional formula ψ must keep being satisfied until condition γ' is met. For example, if a robot is moving forward and there is no obstacle in front of him, it should continue moving forward until the goal region is reached.

These rules state that we can write any literal in subformulas representing conditions (subformulas in the left of an implies or until connective) but, in all other subformulas, we can only require that a given state description symbol maintains its truth value (but cannot enforce the occurrence of an event that changes it) or that a given controllable event cannot occur. Hence, formulas of these types ensure admissibility of the supervisor because we never enforce the following:

- A change in the marking of the system;
- Keeping a token in a place that is in the preset of an uncontrollable transition, and not in its postset;
- The occurrence of an event or set of events (we only enforce that a given subset of E_c cannot occur).

These 3 types of formulas allow us to specify a wide array of behaviours for a system to fulfil. In fact, in our experience, all the natural language specifications we defined for several different examples can be written in this form.

V. Example

We present a direct adaptation of the FSA-based soccer robot example presented in [11]. Consider a soccer team of n robots. The goal is to reach a situation in which one of the robots is close enough to the goal to shoot and score. When a robot does not have the ball in its possession, it can move to the ball until it is close enough to take its possession or get ready to receive a pass from a teammate. When it has the ball, it can shoot the ball, take the ball to the goal if there is no opponent blocking its path or choose a teammate to pass the ball and, when the teammate is ready to receive, pass it. For simplicity, we assume that when a robot shoots the ball the team loses its possession, and that the opponents are only able to block paths. In Fig. 1, we present the PN G_i for robot i. It is a direct translation to a PN of the FSA model in [11].

For each robot i, $D_i = \{moving2getball_i,\ has\_ball_i\}$. The events $close\_to\_ball_i$, $close\_to\_goal_i$ and $blocked\_path_i$ are caused by changes in the environment, hence are uncontrollable. The remaining events are controllable and correspond to the actions available to each robot. For each robot i, we define the set E_c^i as the set of controllable events that can be issued by i, i.e., the set of controllable events of G_i. This set is used to guarantee the supervisor admissibility: instead of writing that a controllable event $e \in E_c^i$ must occur, we write that all other controllable events in E_c^i cannot occur until the occurrence of e. We also add the facts that when a robot is going to the ball it does not have it in its possession and when a robot has the ball, it is not going towards it:

$$K = \{(moving2getball_i, \neg has\_ball_i), (has\_ball_i, \neg moving2getball_i) \mid i = 1,...,n\}$$

A PN model for the whole team is given by the parallel composition of the PNs for each robot, where we synchronize transitions with shared event labels and keep the state description (thus obtaining a PN that models a run in parallel of the individual PNs). The following specifications are used to coordinate the team:

- For the whole team, if a robot is moving to the ball or has the ball in its possession, then no other robot should move to the ball:

$$\varphi = G((\bigvee_{i=1}^{n}(moving2getball_i \vee has\_ball_i)) \Rightarrow (X(\bigwedge_{i=1}^{n} \neg move\_to\_ball_i)))$$

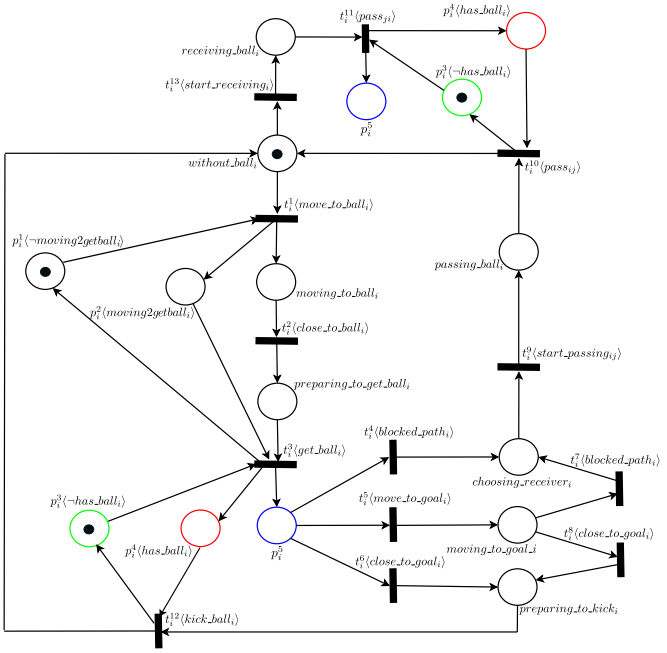
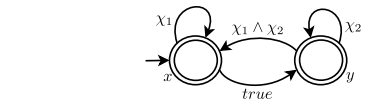


Figure 1. PN model for robot i. Places with the same color represent the same place, we separated them to improve readability. Transition labels and propositional descriptions are depicted as $\langle . \rangle$, e.g., $\mu(\neg has\_ball_i) = p_i^3$ and $\ell(t_i^1) = move\_to\_ball_i$. The names of places that do not correspond to state description symbols are merely used to help the understanding of their meaning and are not used in the method. The transition names are superscripted with an identification number and subscripted with the robot number. Transitions labelled with events $start\_passing_{ij}$, $pass_{ij}$ and $pass_{ji}$, $i \neq j$ are representing $n-1$ transitions, one for each teammate. Furthermore, events $pass_{ij}$, $pass_{ji}$, $i \neq j$ are shared by robots i and j.



$\chi_1 = \neg moving2getball_1 \wedge \neg moving2getball_2 \wedge \neg has\_ball_1 \wedge \neg has\_ball_2$
$\chi_2 = \neg move\_to\_ball_1 \wedge \neg move\_to\_ball_2$

Figure 2. The BA for formula φ

- For each robot i, it will not get ready to receive a pass if none of its teammates wants to pass it the ball:

$$\psi_i = G((\bigwedge_{\substack{j=1 \\ j \neq i}}^{n} \neg start\_passing_{j,i}) \Rightarrow (X \neg start\_receiving_i))$$

- For each robot i, when one of the teammates decides to pass it the ball, it will be ready to receive the pass as soon as possible:

$$\gamma_i = G((\bigvee_{\substack{j=1 \\ j \neq i}}^{n} start\_passing_{j,i}) \Rightarrow (X((\bigwedge_{e' \in E_c^i \setminus \{start\_receiving_i\}} \neg e') U start\_receiving_i)))$$

In Fig. 2, we show the BA B_φ corresponding to formula φ, with $n = 2$. To illustrate the composition algorithm, we will analyse in which conditions can transition t_1^2 fire when the BA is on state x. The valuation generated by the firing of t_1^2 is:

$$v_{t_1^2}(\pi) = \begin{cases} 1 & \text{if } \pi = close\_to\_ball_1 \\ 0 & \text{if } \pi \in E \setminus \{close\_to\_ball_1\} \\ \downarrow & \text{if } \pi \in D \end{cases}$$

Note that $v_{t_1^2}$ is undefined for all $d \in D$ because there is no place in P_D in the postset of t_1^2. Also, in this case, K does not augment the information given by the valuation. For the BA transition labels from x, we have that:

$$[\![\chi_1]\!]^K_{v_{t_1^2}} = \chi_1 \qquad [\![true]\!]^K_{v_{t_1^2}} = true$$

Thus, we add x to the set $next\_possible\_states$ and y to the set $next\_guaranteed\_states$. From this evaluation, we conclude that whenever t_1^2 fires and the observer is in state $\{x\}$, it will evolve to a state that contains $\{y\}$. Now, we need to check, for all combinations of subsets of the set $next\_possible\_states$, in which conditions does the firing of t_1^2 drive us to that subset. Since $next\_possible\_states$ is a singleton, we only have 2 cases:

- t_1^2 *fires and we are guaranteed to go to state* $\{y\}$. To guarantee that we are in this case, we find in which conditions can t_1^2 fire while not satisfying $[\![\chi_1]\!]^K_{v_{t_1^2}}$, i.e., satisfying:

$$\neg\chi_1 = moving2getball_1 \vee moving2getball_2 \vee has\_ball_1 \vee has\_ball_2$$

  Hence, this case yields the construction of four transitions (one for each conjunctive clause). Transition $t_1^{2'}$ in Fig. 3 is the transition obtained for has_ball_1. Note that $t_1^{2'}$ has arcs equal to t_1^2, plus arcs representing the evolution of the observer from $\{x\}$ to $\{y\}$, plus a reflexive-arc to the place representing that has_ball_1 is $true$;
- t_1^2 *fires and we are guaranteed to go to state* $\{x,y\}$. To guarantee that we are in this case, we find in which conditions can t_1^2 fire while satisfying $[\![\chi_1]\!]^K_{v_{t_1^2}}$, i.e., satisfying:

$$\neg moving2getball_1 \wedge \neg moving2getball_2 \wedge \neg has\_ball_1 \wedge \neg has\_ball_2$$

  Hence, this case yields the construction of one transition, because we only obtained one conjunctive clause. This transition is depicted in Fig. 3 as $t_1^{2''}$.

We will build $2n+1$ supervisors, one for formula φ, which deals with the team as a whole, one for each formula ψ_i and one for each formula γ_i. Fig. 4 shows the sum of the sizes (defined as the sum of the number of places and transitions) of the $2n+1$ supervisors, before and after deleting the dead transitions and ranging from a team of 2 to a team of 10 robots. It also shows the size (defined as the number of states) of the modular FSA supervisors obtained using [11]. It is clear that building PN supervisors is a better approach than building FSA supervisors, which scale much more poorly. In fact, we were only able to obtain results for 5 or less robots. Also, we can see that in spite of not using a complete method to delete dead

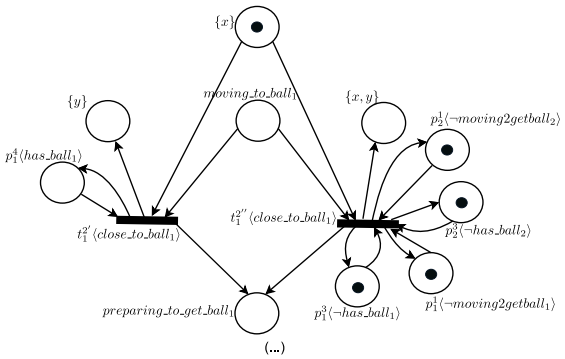
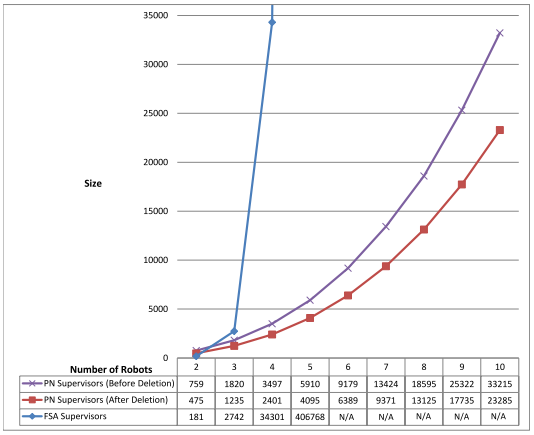


Figure 3. A fragment of the obtained supervisor



Number of Robots	2	3	4	5	6	7	8	9	10
PN Supervisors (Before Deletion)	759	1820	3497	5910	9179	13424	18595	25322	33215
PN Supervisors (After Deletion)	475	1235	2401	4095	6389	9371	13125	17735	23285
FSA Supervisors	181	2742	34301	406768	N/A	N/A	N/A	N/A	N/A

Figure 4. Size of the supervisors – FSA and PN

transitions, we are able to reduce the size of the supervisors. Even though the supervisors are large, we were able to build them in a decent amount of time and for an already large number of robots[5]. We argue that without a formal method that automatically guarantees that the specifications are met, the construction of supervisors for this number of robots would be much more error-prone.

VI. Conclusion and Further Work

This work presents a method to build PN supervisors that are guaranteed to fulfil LTL specifications, presenting the designer with a framework where both the system and the specifications for it to fulfil are represented in suitable formalisms which allow for the implementation of complex tasks. As illustrated in an application example, the method is especially well suited for multi-robot tasks, where the individual behaviours for each robot are modelled as a PNs and the coordination rules between them are specified in LTL.

In the future, we intend to define a decentralized version of the method where we only add to the model of each robot the places and transitions of other robots that are related to the specifications, thus reducing the size of the supervisors. We also plan to address failures in actions and sensor readings, by adding uncertainty to the state description. Furthermore, we plan to formally verify liveness properties of our supervisors, which cannot be implemented using this methodology.

[5]For 10 robots, the supervisors were built in around 2h30m, using an Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz processor and 4GB of RAM.

References

[1] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] H. Costelha and P. Lima. Modeling, analysis and execution of robotic tasks using Petri nets. In *Proc. of IROS '07 – IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1449–1454, San Diego, CA, USA, 2007.

[3] E. A. Emerson. *Temporal and modal logic. In Handbook of theoretical computer science (vol. B)*, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[4] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proc. of CAV '01: 13th Int. Conf. on Comp. Aided Verification*, pages 53–65, London, UK, 2001.

[5] A. Giua and F. DiCesare. Blocking and controllability of Petri nets in supervisory control. *IEEE Transactions on Automatic Control*, 39(4):818–823, 1994.

[6] L. E. Holloway, B. H. Krogh, and A. Giua. A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems*, 7(2):151–190, 1997.

[7] M. V. Iordache and P. J. Antsaklis. Supervision based on place invariants: A survey. *Discrete Event Dynamic Systems*, 16(4):451–492, 2006.

[8] S. Jiang and R. Kumar. Supervisory control of discrete event systems with CTL* temporal logic specifications. *SIAM Journal on Control and Optimization*, 44(6):2079–2103, 2006.

[9] M. Kloetzer and C. Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transactions on Automatic Control*, 53(1):287–297, 2008.

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

[11] B. Lacerda and P. Lima. LTL plan specification for robotic tasks modelled as finite state automata. In *Proc. of Workshop ADAPT – Agent Design: Advancing from Practice to Theory, Workshop at AAMAS '09*, Budapest, Hungary, 2009.

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

[13] M. Pistore and P. Traverso. Planning as model checking extended goals in non-deterministic domains. In *Proceedings of IJCAI '01: 17th Int. Joint Conf. On Artificial Intelligence*, pages 479–484, Seattle, WA, USA, 2001.

[14] W. Sheng and Q. Yang. Peer-to-peer multi-robot coordination algorithms: Petri net based analysis and design. In *Proc. of IEEE/ASME '05: The 2005 Int. Conf. on Advanced Intelligent Mechatronics*, pages 1407–1412, Monterey, CA, USA, 2005.