Linear-Time Temporal Logic Control of Discrete Event Models of Cooperative Robots

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

***Abstract*—A Discrete Event System (DES) is a discrete state space dynamic system that evolves in accordance with the instantaneous occurrence, at possibly unkown times, of physical events. Given a DES, its behavior (the sequence of displayed events) may not satisfy a set of logical performance objectives. The purpose of Supervisory Control is to restrict that behavior in order to achieve those objectives. Linear-Time Temporal Logic (LTL) is an extension of Propositional Logic which allows reasoning over an infinite sequence of states. We will use this logical formalism as a way to specify our performance objectives for a given DES and build a supervisor that restricts the DES' behavior to those objectives by construction. Several simulated application examples illustrate the developed method.**

***Index Terms*—Discrete Event Systems, Supervisory Control, Linear-Time Temporal Logic**

I. Introduction

In recent years there has been a considerable interest in Discrete Event Systems (DES), whose discrete states change in response to the occurrence of events from a predefined event set. Examples of such systems can be found in communication networks, computer programs, operating systems, manufacturing processes and robotics. One of the main fields of study is Supervisory Control, introduced in [9] and further developed in [2], which focuses on the restriction of a DES behavior in order to satisfy a set of performance objectives. This restriction is, in many cases, performed in an *ad-hoc* manner, but with the continuing growth of this type of systems, a more generalized framework is needed. In this work, we present a framework to restrict a DES behavior specifying its performance objectives with Linear-Time Temporal Logic (LTL). Using this approach, we guarantee that the required behavior is achieved by construction. Furthermore, in many cases, the specification of the performance objectives using LTL is almost immediate, allowing the supervision of more complex systems. A great deal of work has been done recently in a slightly different context: controlling continuous state space time-driven linear systems with LTL specifications ([1], [7], [10]). In this context, a discretization of the linear system is needed before the LTL specification can be enforced, obtaining a discrete space system. The system is then refined to a hybrid system. This approach is mainly used to perform robot motion planning (enforcing a robot to go to certain places and avoid certain obstacles). Our approach is different because we will be concerned with, given a team of robots where we assume that each one can perform a number of tasks individually, coordinating their behavior so that they reach a given objective. For this purpose DES models are more suitable and reduce the involved complexity by comparison to hybrid systems models. LTL enables the formulation of complex sentences by compact logical sentences.
The work in divided in three main Sections: In Section 2 we introduce the notions of Discrete Event System and Supervisory Control, explaining how one can see a Finite State Automaton as a DES. In Section 3 we define Linear-Time Temporal Logic and mention a method to build a Büchi automaton that accepts exactly the ω-language of the infinite sequences that satisfy a given formula φ. Finally, in Section 4 we congregate all the theory defined throughout this work to present our method of supervisory control and give some operational examples of applications of the presented method. The developed approach is illustrated with simulation examples that are deployed along the paper.

II. Discrete Event Systems

A. Preliminaries

Definition 1 (Discrete Event System): A Discrete Event System is composed of a discrete set X of possible states and a finite set $E = \{e_1, ..., e_m\}$ of possible events.
At a given time $t \geq 0$, the DES is in a given state $x \in X$, which is all the information needed to characterize the system at that time instant. The state of a DES can only be changed by the occurence of an event $e \in E$ and these events occur both instanteneously and asynchronously.

The set X is called the *state-space* of the DES and the set E is called the *event-space* of the DES. Both these sets must be discrete and E must be finite. We can interpret the state as the task the system if performing at a given moment, such as a robot moving forward, a machine being idle or a computer running a program. The events are interpreted as physical phenomenons, such as a robot's sensor detecting something, a new job arriving to a machine or a program crashing.

Example 1 (Transporting robots): Consider two robots, each one holding one end of a bar. Their objective is to transport the bar to another place. To simplify, assume that the robots can only move a constant distance forward or stop.

This situation can be modeled as a DES with $X = \{$Both robots stopped, Robot$_1$ moving and Robot$_2$ stopped, Robot$_1$ stopped and Robot$_2$ moving, Both robots moving$\}$ and $E = \{Move_1, Move_2, Stop_1, Stop_2\}$.
A sequence of events in this DES can be $((Move_1, t_1), (Stop_1, t_2), (Move_1, t_3), (Move_2, t_4), (Stop_1, t_5), (Stop_2, t_6))$, $t_1 < t_2 < ... < t_6$.

In this example, one of the robots can move forward to a position where it is too far from the other one, making the bar fall.

B. Modeling Logical DES

There are three levels of abstraction usually considered in the study of DES, *Untimed (or logical) DES models*, *Deterministic Timed DES Models* and *Stochastic Timed DES Models*.

The theory of Supervisory Control is defined over Logical DES Models, so in this work we will introduce Finite State Automata as our modeling framework.

Definition 2 (Finite State Automaton): A Finite State Automaton (FSA) is a six-tuple $G = (X, E, f, \Gamma, X_0, X_m)$ where:

- X is the finite set of states
- E is the finite state of events
- $f : X \times E \rightarrow X$ (deterministic) or $f : X \times E \rightarrow 2^X$ (nondeterministic) is the (possibly partial) transition function
- $\Gamma : X \rightarrow 2^E$ is the active event function
- $X_0 \subseteq X$ is the initial state (a singleton for deterministic FSA)
- $X_m \subseteq X$ is the set of marked states

Deterministic FSA (DFA) and Nondeterministic FSA (NFA) are equivalent, as proven in [5]. The following definitions will be made for DFA, but the generalization for NFA is straightforward. $f(x, e) = y$ means that there is a transition labeled by event e from state x to state y. $\Gamma(x)$ is the set of all events e for which $f(x, e)$ is defined. Note that Γ is uniquely defined by f, it was included in the definition for convenience. We also extend f from domain $X \times E$ to domain $X \times E^*$ in the following recursive manner:

- $f(x, \epsilon) = x$
- $f(x, se) = f(f(x, s), e),\ s \in E^*,\ e \in E$

Now, we are in conditions to define the languages generated and marked by a DFA. As we will see, the objective of Supervisory Control is to restrict these languages to the strings we consider "legal" for our system.

Definition 3 (Generated and Marked Languages): Let $G = (X, E, f, \Gamma, x_0, X_m)$ be a DFA. We define

- $L(G) = \{s \in E^* : f(x_0, s)\ is\ defined\}$, the language generated by G
- $L_m(G) = \{s \in L(G) : f(x_0, s) \in X_m\}$, the language marked by G

The notion of marked language is used to model "complete tasks" of a DES. We will now introduce three operations over DFA that are very useful in DES modeling and necessary to perform supervision.

Definition 4 (Acessible Part): Let $G = (X, E, f, \Gamma, x_0, X_m)$ be a DFA. The accessible part of G is the DFA $Ac(G) = (X_{ac}, E, f_{ac}, \Gamma_{ac}, x_0, X_{ac,m})$ where

- $X_{ac} = \{x \in X : \exists s \in E^* f(x_0, s) = x\}$
- $X_{ac,m} = X_m \cap X_{ac}$
- $f_{ac} = f|_{X_{ac} \times E \rightarrow X_{ac}}$
- $\Gamma_{ac} = \Gamma|_{X_{ac} \rightarrow 2^E}$

The accessible part of a DFA is simply its restriction to the states that can be reached from the initial state. f_{ac} is the restriction of f to domain $X_{ac} \times E$ and Γ_{ac} is the restriction of Γ to domain X_{ac}. It is clear that $L(G) = L(Ac(G))$ and $L_m(G) = L_m(Ac(G))$.

Definition 5 (Product Composition): Let $G_1 = (X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1})$ and $G_2 = (X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2})$ be two DFA. The product composition of G_1 and G_2 is the DFA $G_1 \times G_2 = Ac(X_1 \times X_2, E_1 \cap E_2, f, \Gamma_{1\times2}, (x_{01}, x_{02}), (X_{m1} \times X_{m2}))$ where

$$f((x_1, x_2), e) = \begin{cases} (f_1(x_1), f_2(x_2)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

and thus $\Gamma_{1\times2}(x_1, x_2) = \Gamma_1(x_1) \cap \Gamma_2(x_2)$

The product composition is also called the *completely synchronous composition*. In this composition, the transitions of the two DFA must always be synchronized on a common event $e \in E_1 \cap E_2$. This means that an event occurs in $G_1 \times G_2$ if and only if it occurs in both DFA. Thus, it is easily verified that $L(G_1 \times G_2) = L(G_1) \cap L(G_2)$ and $L_m(G_1 \times G_2) = L_m(G_1) \cap L_m(G_2)$.

Definition 6 (Parallel Composition): Let $G_1 = (X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1})$ and $G_2 = (X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2})$ be two DFA. The parallel composition of G_1 and G_2 is the DFA $G_1 \parallel G_2 = Ac(X_1 \times X_2, E_1 \cup E_2, f, \Gamma_{1\parallel2}, (x_{01}, x_{02}), (X_{m1} \times X_{m2}))$ where

$$f((x_1, x_2), e) = \begin{cases} (f_1(x_1), f_2(x_2)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1), x_2) & \text{if } e \in \Gamma_1(x_1) \setminus E_2 \\ (x_1, f_2(x_2)) & \text{if } e \in \Gamma_2(x_2) \setminus E_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and thus $\Gamma_{1\parallel2}(x_1, x_2) = [\Gamma(x_1) \cap \Gamma(x_2)] \cup [\Gamma(x_1) \setminus E_2] \cup [\Gamma(x_2) \setminus E_1]$

The parallel composition is also called the *synchronous composition*. In this composition, an event in $E_1 \cap E_2$ (common event) can only be executed if the two DFA both execute it sinultaneously. An event in $(E_2 \setminus E_1) \cup (E_1 \setminus E_2)$ (private event) can be executed whenever possible. If $E_1 = E_2$, then the parallel composition reduces to the product, since all transitions must be synchronized and if $E_1 \cap E_2 = \emptyset$, then there are no synchronized transitions and $G_1 \parallel G_2$ models the concurrent behavior of G_1 and G_2 (in this case we call $G_1 \parallel G_2$ the *shuffle* of G_1 and G_2).

Example 2 (Transporting Robots): The DES of Example 1 can be modeled by the FSA shown in Figure 1.

Another way of modeling this system is using parallel composition, which is very useful when our system has several components operating concurrently. It allows us to model each component separately and then get the FSA that

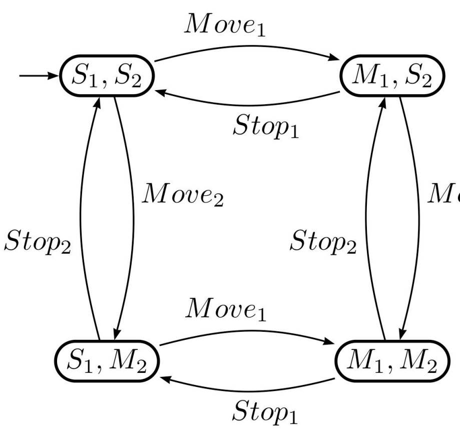

Fig. 1. FSA model of Transporting Robots

models the whole system by applying it. Hence, if we model each robot separately, we obtain the FSA $G_i, i = 1, 2$, seen in Figure 2.

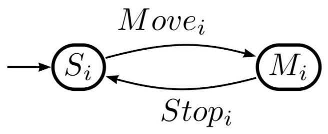

Fig. 2. FSA model of Transporting Robot i

It is easy to see that $G_1 \parallel G_2$ is the FSA represented in Figure 1.

Example 3 (Robotic Soccer): Consider a team of n robots playing a soccer game. The objective 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 has two options:

- Move to the ball until it is close enough to take its possession, or
- Get ready to receive a pass from a teammate.

When a robot has the possession of the ball, it can:

- Shoot the ball (if it is close enough to the goal), or
- Take the ball to the goal, if there is no opponent blocking its path, or
- Choose a teammate to pass the ball and, when it is ready. to receive, pass it

For simplicity, we assume that, when a robot shoots the ball, the team loses its possession (we do not differentiate the situation where the robot scores from the situation where the robot does not score since the team will lose the ball's possession in both) and that the opponents do not steal the ball (they are only able to block paths, at which point our robot will try to pass to a teammate). Figure 3 depicts a possible FSA R_i model for robot i. An FSA model for the whole team is given by $T = R_1 \parallel R_2 \parallel ... \parallel R_n$. Note that the $pass(i, j)$ event must be synchronized between robot i (the passing robot) and robot j (the receiving robot).

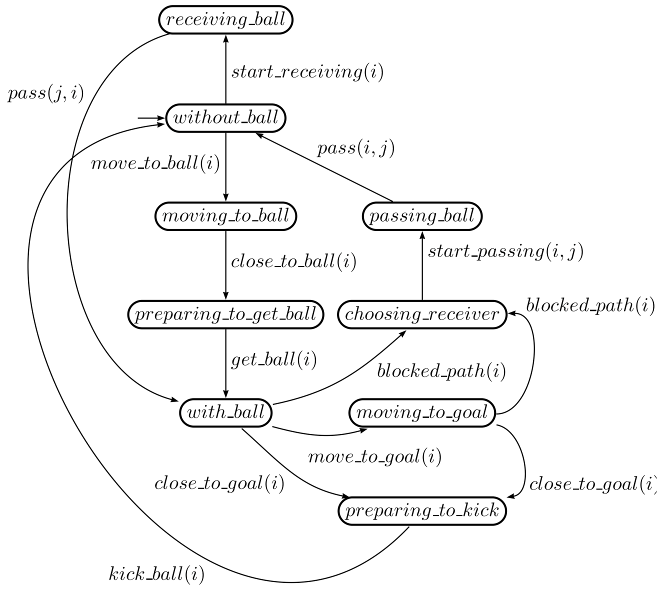

Fig. 3. FSA for Robot R_i

Note that, when we write $start\_passing(i, j)$, $pass(i, j)$ and $pass(j, i)$ in a transition, we are representing $n-1$ events, since $j = 1, ..., n,\ j \neq i$.

III. Supervisory Control

As we have seen in previous examples, sometimes our DES model has some behaviors that are not satisfactory. Let's assume we have a DES modeled by FSA G. G models the "uncontrolled behavior" of the DES and is called the *plant*. Our objective is to modify the plant's behavior, i.e., restrict its behavior to an admissible language $L_a \subseteq L(G)$, using control.

To do this, we start by partitioning the event set E in two disjoint subsets $E = E_c \cup E_{uc}$.

E_c is the set of *controllable events*, i. e., the events that can be prevented from happening and E_{uc} is the set of *uncontrollable events* , i.e., the events that cannot be prevented from happening. This partition is due to the fact that, in general, there are events that make a DES change its state that are not of the "responsibility" of the DES itself.

Example 4: We list the set of controlled and uncontrolled events in previous examples.

- In Example 2, we assume that the robots can only move a constant distance forward. Hence, after a robot starts moving, the decision to stop is not its responsibility, it always stops after it moves the predefined distance.
  - $E_c = \{Move_1,\ Move_2\}$
  - $E_{uc} = \{Stop_1,\ Stop_2\}$
- In Example 3 the events $close\_to\_ball$, $close\_to\_goal$ and $blocked\_path$ are caused by changes in the environment around the robots and not by the robots themselves.

Therefore, they are considered uncontrollable events. The controllable events correspond to the actions available to each robot.

- $E_c = \{move\_to\_ball(i),\ get\_ball(i),\ kick\_ball(i),\ move\_to\_goal(i),\ start\_passing(i,j),\ start\_receiving(i),\ pass(i,j) : i,j = 1,...,n,\ j \neq i\}$
- $E_{uc} = \{close\_to\_ball(i),\ blocked\_path(i),\ close\_to\_goal(i) : i = 1,...,n\}$

Next, we introduce the notion of a DES $G = (X, E = E_c \cup E_{uc}, f, \Gamma, X_0, X_m)$ controlled by a supervisor S. Formally, a supervisor is a function $S : L(G) \rightarrow 2^E$ that, given $s \in L(G)$ outputs the set of events G can execute next (*enabled events*). We only allow supervisors S such that, when event $e \in E_{uc}$ is active in the plant G, it is also enabled by S. That is, a supervisor must always allow the plant to execute its uncontrollable events.

Definition 7 (Admissible Supervisor): Let $G = (X, E = E_c \cup E_{uc}, f, \Gamma, x_0, X_m)$ be a DES and $S : L(G) \rightarrow 2^E$. S is an admissible supervisor for G if for all $s \in L(G)$ $E_{uc} \cap \Gamma(f(x_0, s)) \subseteq S(s)$.

We will check the admissibility of our supervisors S in a case-by-case basis.

Definition 8 (Controlled DES): Let $G = (X, E = E_c \cup E_{uc}, f, \Gamma, x_0, X_m)$ be a DES and $S : L(G) \rightarrow 2^E$. The controlled DES (CDES) S/G (S controlling G) is a DES that constrains G in such a way that, after generating a string $s \in L(G)$, the set of events that S/G can execute next (enabled events) is $S(s) \cap \Gamma(f(x_0, s))$.

The way S/G operates is represented in Figure 4 and is as follws: s is the string of all events executed so far by G, which is observed by S. S uses s to determine what events should be enabled, that is, which events can occur after the generation of s.

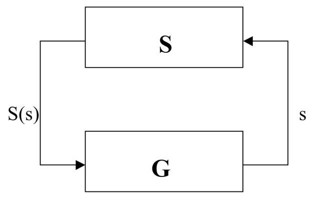

Fig. 4. The feedback loop of supervisory control

Definition 9 (Generated and Marked Languages by a CDES): Let S/G be a CDES and e one of its events. The language generated by S/G, $L(S/G)$, is defined as follows:

- $\epsilon \in L(S/G)$;
- if $s \in L(G)$ and $se \in L(G)$ and $e \in S(s)$ then $se \in L(S/G)$.

and the language marked by S/G, $L_m(S/G)$, is

- $L_m(S/G) = L(S/G) \cap L_m(G)$.

Thus, given a plant G and an admissible language $L_a \subseteq L(G)$, we want to find a supervisor S such that $L(S/G) = L_a$ (in this work we will be focused on generated languages and will not be concerned with marked languages).

In this framework, the supervisor is usually implemented by an FSA R, such that $L(R) = L_a$. R is refered to as the *standard realization* of S. The most common method to build R is to start by building a simple FSA H_{spec} that captures the essence of the natural language specification and then combine it with G, using either product or parallel composition. We choose parallel composition if the events that appear in G but not in H_{spec} are irrelevant to the specification that H_{spec} implements or product composition when, on the other hand, the events that appear in G but not in H_{spec} should not happen in the admissible behavior L_a.

Having the FSA $G = (X_G, E_G, f_G, \Gamma_G, x_{G,0}, X_{G,m})$ and $R = (X_R, E_R, f_R, \Gamma_R, x_{R,0}, X_{R,m})$ that represent the plant and the standard realization of S respectively (note that $E_R \subseteq E_G$), the feedback loop of supervisory control is implemented as follows: Let G be in state x and R be in state y following the execution of string $s \in L(S/G)$. G executes an event e that is currently enabled, i.e., $e \in \Gamma_G(x) \cap \Gamma_R(y)$. R also executes the event, as a passive observer of G. Let $x' = f_G(x, e)$ and $y' = f_R(y, e)$ be the new states of G and R respectively, after the execution of e. The set of enabled events of G after string se is now given by $\Gamma_G(x') \cap \Gamma_R(y')$. It is common to make $X_{R,m} = X_R$, so that $R \times G$ represents the closed-loop system S/G:

- $L(R \times G) = L(R) \cap L(G) = L_a \cap L(G) = L_a = L(S/G)$
- $L_m(R \times G) = L_m(R) \cap L_m(G) = L_a \cap L_m(G) = L(S/G) \cap L_m(G) = L_m(S/G)$

So, from now on, we will refer to a supervisor S and its standard realization R interchangeably.

Next, we address modular supervision, a mean of reducing the complexity of the controlled DES model.

Definition 10 (Modular Supervision): Let $S_1, ..., S_n,\ n \in \mathbb{N}$ be admissible supervisors for DES $G = (X, E = E_c \cup E_{uc}, f, \Gamma, x_0, X_m)$ and $s \in L(G)$. We define the (admissible) modular supervisor as

- $S_{mod12...n}(s) = S_1(s) \cap S_2(s) \cap ... \cap S_n(s)$

It is obvious, by definition 7 that $S_{mod12...n}$ is admissible for G. In Figure 5 we represent modular supervision with 2 supervisors. In modular control, an event is enabled by $S_{mod12...n}$ if and only if it is enabled for all $S_i,\ i = 1, ..., n$.

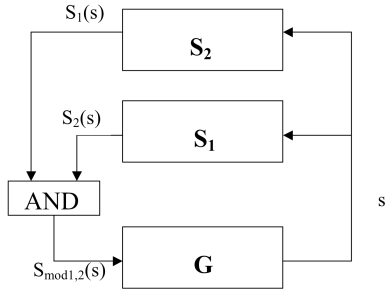

Fig. 5. The feedback loop of modular supervisory control

Remark 1 (Multiple Specifications): When our admissible behavior is composed of multiple specifications, that is, when $L_a = L_{a,1} \cap ... \cap L_{a,n}$, where $L_{a,i}$ represents a given specification we want our plant G to satisfy, we will build n supervisors $S_i,\ i = 1, ..., n$ such that $L(S_i/G) = L_{a,i}$ and use modular control to implement a supervisor $S_{mod1...n}$ such that $L(S_{mod1...n}/G) = L_a$.

Example 5 (Transporting Robots): As we have mentioned, it is possible for one robot to move forward to a position where it is too far from the other, making the bar fall. One way to avoid this is to impose alternation between the robots' motion: one robot moves forward while the other is stopped, holding the bar. Then the other robot moves forward while the one that moved before is stopped, holding the bar, etc. So, we have 4 specifications:

- **Spec** 1 **-** Robot 1 cannot start moving while Robot 2 is moving;
- **Spec** 2 **-** Robot 2 cannot start moving while Robot 1 is moving;
- **Spec** 3 **-** After Robot 1 moves, it will only start moving again after Robot 2 has moved;
- **Spec** 4 **-** After Robot 2 moves, it will only start moving again after Robot 1 has moved.

Example 6 (Robotic Soccer): Regarding Example 3, one may define the following specifications, which are useful to improve the team's performance in a soccer game for each Robot i:

- **Spec** $1, i$ **-** If another teammate goes to the ball, robot i will not go to the ball until it is kicked by some robot in the team;
- **Spec** $2, i$ **-** Robot i will not get ready to receive a pass, unless one of its teammates decides to pass it the ball and, in this case, it will be ready to receive the pass as soon as possible.

Spec $1, i$ guarantees that only one robot moves to the ball at a time and that, when the team has the ball, no robot moves to it and Spec $2, i$ guarantees that no robot will be ready to receive a pass when none of its teammates wants it to receive a pass and that when a robot wants to pass the ball, another one will get ready to receive it as soon as possible .

IV. Linear-Time Temporal Logic and Büchi Automata

In this Section we introduce Linear-Time Temporal Logic (LTL). We start by defining the syntax and semantics of LTL and then refer the translation from LTL formulas to Büchi Automata.

A. Linear-Time Temporal Logic

LTL is an extension of Propositional Logic which allows reasoning over an infinite sequence of states. LTL is widely used for verification of properties of several concurrent systems (for example, *safety* and *liveness*), especially software systems. In the following, Π is a set of propositional symbols.

Definition 11 (Syntax): The set $L_{LTL}(\Pi)$ of LTL formulas over Π is defined inductively as follows:

- $true,\ false \in L_{LTL}(\Pi)$;
- If $p \in \Pi$ then $p \in L_{LTL}(\Pi)$;
- If $\varphi, \psi \in L_{LTL}(\Pi)$ then $(\neg\varphi), (\varphi \vee \psi), (\varphi \wedge \psi) \in L_{LTL}(\Pi)$;
- If $\varphi \in L_{LTL}(\Pi)$ then $(X\varphi) \in L_{LTL}(\Pi)$;
- If $\varphi, \psi \in L_{LTL}(\Pi)$ then $(\varphi U \psi) \in L_{LTL}(\Pi)$;
- If $\varphi, \psi \in L_{LTL}(\Pi)$ then $(\varphi R \psi) \in L_{LTL}(\Pi)$.

In Definitions 12 and 13, we define the LTL semantics.

Definition 12 (Local Satisfaction): Let $\sigma : \mathbb{N} \to 2^{\Pi}$, $t \in \mathbb{N}, p \in \Pi$ and $\varphi, \psi \in L_{LTL}(\Pi)$. The notion of satisfaction ($\Vdash$) is defined as follows:

- $\sigma(t) \Vdash true$ and $\sigma(t) \nVdash false$;
- $\sigma(t) \Vdash p$ if and only if $p \in \sigma(t)$;
- $\sigma(t) \Vdash (\neg\varphi)$ if and only if $\sigma(t) \nVdash \varphi$;
- $\sigma(t) \Vdash (\varphi \vee \psi)$ if and only if $\sigma(t) \Vdash \varphi$ or $\sigma(t) \Vdash \psi$;
- $\sigma(t) \Vdash (\varphi \wedge \psi)$ if and only if $\sigma(t) \Vdash \varphi$ and $\sigma(t) \Vdash \psi$;
- $\sigma(t) \Vdash (X\varphi)$ if and only if $\sigma(t+1) \Vdash \varphi$;
- $\sigma(t) \Vdash (\varphi U \psi)$ if and only if exists $t' \geq t$ such that $\sigma(t') \Vdash \psi$ and for all $t'' \in [t, t'[\ \sigma(t'') \Vdash \varphi$;
- $\sigma(t) \Vdash (\varphi R \psi)$ if and only if for all $t' \geq t$ such that $\sigma(t') \nVdash \psi$ exists $t'' \in [t, t'[$ such that $\sigma(t'') \Vdash \varphi$.

Definition 13 (Global Satisfaction): Let $\sigma : \mathbb{N} \to 2^{\Pi}$ and $\varphi \in L_{LTL}(\Pi)$. The notion of global satisfaction is defined as follows:

- $\sigma \Vdash \varphi$ if and only if $\sigma(0) \Vdash \varphi$.

Now, we give a brief explanation of each operator defined:

- The X operator is read "next", meaning that the formula it precedes will be true in the next state;
- The operator U is read "until", meaning that its first argument will be true until its second argument becomes true (and the second argument must become true in some state, i.e., an ω-string where φ is always satisfied but ψ is never satisfied does not satisfy $\varphi U \psi$);
- The operator R, which is the dual of U, is read "releases", meaning that its second argument must always be true until its first argument becomes true (in this case, an ω-string where ψ is always satisfied satisfies $\varphi R \psi$, because the definition does not require the existence of t').

There are two other commonly used temporal operators, F and G, usually defined by abbreviation.

Definition 14 (Abbreviations): Let $p \in \Pi$ and $\varphi, \psi \in L_{LTL}(\Pi)$. We define the following abbreviations:

- $(\varphi \Rightarrow \psi) \equiv_{abv} ((\neg\varphi) \vee \psi)$;
- $(\varphi \Leftrightarrow \psi) \equiv_{abv} ((\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi))$;
- $(F\varphi) \equiv_{abv} (trueU\varphi)$;
- $(G\varphi) \equiv_{abv} (falseR\varphi)$.

- The F operator is read "eventually", meaning that the formula it precedes will be true in a future state;
- The G operator is read "always", meaning the formula it precedes will be true in all future states.

B. Büchi Automata

Büchi Automata are used to describe ω-languages, i.e., languages of infinite strings[1] (ω-strings). Büchi automata have

[1]One should notice that a function $\sigma : \mathbb{N} \to 2^{\Pi}$ is in fact an ω-string whose i-th element is given by $\sigma(i-1)$.

the same structure as FSA, The characteristic that sets them apart is their semantics, since for Büchi Automata one defines generated and marked ω-languages instead of generated and marked languages.

To define the generated and marked ω-languages by a Büchi Automaton, we need to introduce the notion of valid state labeling.

Definition 15 (Valid State Labeling): Let $B = (X, E, f, \Gamma, X_0, X_m)$ be a Büchi automaton and $\sigma \in E^\omega$ an ω-string. A valid state labeling for B and σ is a function $\rho : \mathbb{N} \to X$ such that:

- $\rho(0) \in X_0$
- $\rho(i+1) \in f(\rho(i), \sigma(i)),\ for\ all\ i \in \mathbb{N}$

We denote $P(B, \sigma)$ as the set of all possible valid state labelings for B and σ.

A valid state labeling for B and σ is an ω-string over the state set of B, where $\rho(i)$ is one of the possible states B can be in (in the deterministic case, the state where B is), while applying its transition function to σ_i. If, for some $i \in \mathbb{N}$, event $\sigma(i+1)$ is not active for any of the possible states B can be in, that is,

$$\sigma(i+1) \notin \bigcup_{x_0 \in X_0} \Big(\bigcup_{x \in f(x_0, \sigma_i)} \Gamma(x) \Big)$$

no such function exists.

Definition 16 (Generated ω-Language by Büchi Automata): Let $B = (X, E, f, \Gamma, X_0, X_m)$ be a Büchi automaton. We define the ω-language generated by B as

- $L(B) = \{\sigma \in E^\omega : P(B, \sigma) \neq \emptyset\}$

The generated ω-strings by B are the ones for which there exists a valid state labeling.

Definition 17 (Marked ω-Language by Büchi Automata): Let $B = (X, E, f, \Gamma, X_0, X_m)$ be a Büchi automaton. We define the ω-language marked by B as

- $L_m(B) = \{\sigma \in L(B) : exists\ \rho \in P(B, \sigma)\ such\ that\ inf(\rho) \cap X_m \neq \emptyset\}$

where, for $\chi \in X^\omega$, $inf(\chi) \subseteq X$ is the set of all $x \in X$ that appear infinite times in χ.
The marked ω-strings by B are the ones generated by "runs" of B that visit at least one of the marked states infinite times.

Now, we state the Theorem that allows us to perform Supervisory Control over a DES given a set of LTL formulas stating our performance objectives. The proof of this theorem is constructive and yields a method to construct the Büchi Automaton that marks the sequences that satisfy a given formula φ. [11] presents the most immediate proof of the theorem and [4] describes a most efficient method for the translation, which is used to calculate the examples we will present later.

Theorem 1: Let $\varphi \in L_{LTL}(\Pi)$. Then there exists a (non-deterministic) Büchi automaton B_φ such that

$$\sigma \Vdash \varphi \quad \text{if and only if} \quad \sigma \in L_m(B_\varphi)$$

V. Supervisor Synthesis

In this Section, we explain how to define the LTL - based supervisor for a plant G and a set of LTL formulas $\varphi_1, ..., \varphi_n,\ n \in \mathbb{N}$. As we have seen, the first step in building a standard realization of a supervisor S, such that $L(S/G) = L_a$ is to construct an FSA H_{spec} that captures the essence of our natural language specification. The construction of H_{spec} can be very error - prone and, in general, not obvious. On the other hand, translating natural language to LTL formulas is, in most cases, straightforward. Thus, we can define our performance objectives in LTL and use the Büchi Automaton referred in Theorem 1 to solve our problem in a much more user - friendly way.

Note that, in order to restrict $L(G)$ to L_a, we will be constructing LTL formulas over the set of propositional symbols E (G's event set), i.e., we will be interested in formulas $\varphi \in L_{LTL}(E)$. Since we assume the occurence of events in a DES to be asynchronous, at each state exactly one event can occur. This allows us to assume $\sigma : \mathbb{N} \to E$ in Definition 12 and substitute condition $\sigma(t) \Vdash p$ if and only if $p \in \sigma(t)$ by $\sigma(t) \Vdash e$ if and only if $\sigma(t) = e$, for $t \in \mathbb{N}$ and $e \in E$. Thus, given a Büchi automaton B_φ, we can delete all events that are not singletons in B_φ's event set and redefine B_φ's transition function accordingly.

Since a Büchi automaton's structure is the same as an NFA, we consider B_φ as an NFA. Next, we need to find the equivalent DFA, H_φ, of B_φ. This must be done because, if we build a supervisor from B_φ, it will disable some events that should not be disabled, due to the nondeterministic choices that are made when an event occurs at a given state and there is more than one state we can go to, e.g., if $f(x, e) = \{y, z\}$ we want the enabled events in state $f(x, e)$ to be $\Gamma(y) \cup \Gamma(z)$ but if we nondeterministically jump to state y we will not be enabling the events in $\Gamma(z) \setminus \Gamma(y)$. This problem is solved by using the equivalent DFA, thus keeping track of all the states B_φ can be in and enabling all the events that are active in at least one of those states. As seen in [5], finding the equivalent DFA of an NFA is an exponential operation, but, in general, the LTL formulas that are relevant to perform supervision yield small Büchi automata. Despite that, the complexity issue is a major one when applying this theory, as we will see in the next Section. Then, we obtain the supervisor $S_\varphi = G \parallel H_\varphi$ or $S_\varphi = G \times H_\varphi$, depending on our supervision problem. Using this method, we guarantee that for all $s \in L(S_\varphi/G)$, there exists $\sigma \in E^\omega$ such that $s\sigma \Vdash \varphi$, i.e., the generated language of the CDES S/G is always in conformity with the specification given by φ. Since the generated language by a CDES is a set of finite strings, this is the best we can have in this framework. We can now describe the method we will use for supervision. Given a plant G and a set of formulas $\{\varphi_1, ..., \varphi_n\},\ n \in \mathbb{N}$ representing the specifications we want G to fulfill, we build the supervisors $S_{\varphi_1}, ..., S_{\varphi_n}$, as explained above, and perform modular supervision, as explained in Section III. The use of modular supervision gives us a gain in efficiency ([9]) and, in addition, allows us to translate the formulas $\varphi_1, ..., \varphi_n$ to Büchi automata one by one, which also allows a significant improvement in the efficiency of the method: If $r_1, ..., r_n$ is the size (number of operators) of $\varphi_1, ..., \varphi_n$ respectively, then

- If we had not opted for modular control, to enforce all the specifications given by $\varphi_1, ..., \varphi_n$ we would need to build a Büchi automaton B_φ for formula

– $\varphi = \left(\bigwedge_{i=1}^{n} \varphi_i\right)$

It is easy to see that φ has, at most, size

– $r = (\sum_{i=1}^{n} r_i) + n - 1$

where the $n-1$ factor is due to the $n-1$ "and" ($\wedge$) operators we added to φ. Hence, B_φ would have, at most, the following number of states (we have seen that the translation from an LTL formula to a Büchi automaton yields an automaton whose number of states is exponential in the size of the formula):

– $|B_\varphi| = 2^r$

- Using modular supervision, we need to build n Büchi automata $B_{\varphi_1}, ..., B_{\varphi_n}$, which, altogether, have at most the following total number of states:

– $\sum_{i=1}^{n} |B_{\varphi_i}| = \sum_{i=1}^{n} 2^{r_i}$

which is clearly better than the previous option's worst case scenario.

VI. Examples

In this section, we present some applications of the framework defined throughout this work. We will build supervisors for the DES in Examples 2 and 3 that enforce the specifications we gave in natural language in Examples 5 and 6. To build these examples, some functions were implemented in *Matlab*. These functions can be found in *http://islab.isr.ist.utl.pt/ltldes_examples.zip*:

- A function that receives a NFA and outputs its equivalent DFA;
- A function that receives two FSA and outputs their product composition;
- A function that receives two FSA and outputs their parallel composition;
- A function that receives a set of LTL formulas and translates them to Büchi automata (this function uses the implementation described in [4] to build the Büchi automaton, which is written in *C* and adapts a *Matlab* function written for the implementation described in [7] to take the output of the *C* function and turn it into a viable *Matlab* structure);
- A function that, given a plant and n supervisors, simulates the feedback loop of modular control;
- A function that congregates all of the above. This function receives a plant and n LTL formulas, creates the supervisors and simulates the feedback loop of modular control.

Example 7 (Transporting Robots): Let's return to the transporting robots example and let G be the FSA represented in Example 2. In Example 5 we defined 4 specifications that prevent the robots from moving to a position where they are too far from the other, making the bar fall. Spec i can be translated to LTL by formula φ_i, where

- $\varphi_1 = (G(Move_2 \Rightarrow (X((\neg Move_1) U Stop_2))))$
- $\varphi_2 = (G(Move_1 \Rightarrow (X((\neg Move_2) U Stop_1))))$
- $\varphi_3 = (G(Move_1 \Rightarrow (X((\neg Move_1) U Stop_2))))$
- $\varphi_4 = (G(Move_2 \Rightarrow (X((\neg Move_2) U Stop_1))))$

Looking at these formulas, one can see that the events that can be disabled are $Move_1$ and $Move_2$. Hence, an admissible supervisor will be obtained. We construct the DFA $H_{\varphi_i},\ i = 1,2,3,4$ from the Büchi automata, as explained before. In Figure 6 we represent the Büchi automaton obtained from φ_2. Next, we obtain the 4 supervisors $S_i = G \parallel H_{\varphi_i}$. In Figure 7 we represent the supervisor S_2. Note that the states reached after an event $Move_1$ happens do not have the event $Move_2$ in their active event set. The modular supervisor $S_{mod1234}$ implements the robot alternation. The controlled system only allows 2 types of strings, $Move_1 - Stop_1 - Move_2 - Stop_2 - Move_1 - Stop_1 - Move_2 - Stop_2 - ...$ or $Move_2 - Stop_2 - Move_1 - Stop_1 - Move_2 - Stop_2 - Move_1 - Stop_1 - ....$ In Figure 8, we represent the automaton $G \times S_1 \times S_2 \times S_3 \times S_4$ which, as we have seen, represents the controlled system. One should notice that our controlled system is not minimum, i.e., there is a 5 states DFA that implements the robot alternation. This is one drawback of this method: in general the controlled system is not the smallest it could be.

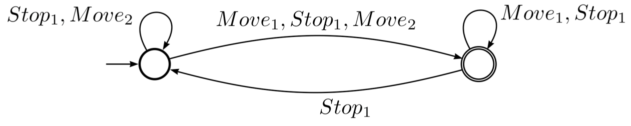

Fig. 6. Büchi automaton marking the ω - strings that satisfy φ_2

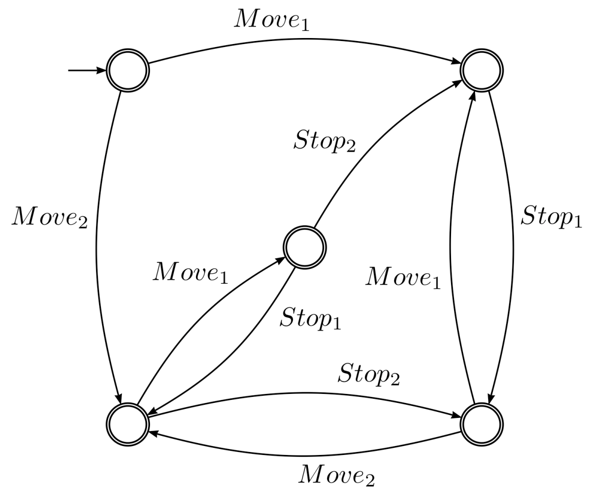

Fig. 7. The supervisor S_2, obtained by formula φ_2

Example 8 (Robotic Soccer): Regarding Example 6, it is easier to represent Spec $1, i,\ i = 1, ..., n$ by only one formula

- $\varphi_1 = (G[(\bigvee_i move\_to\_ball(i))$
$\Rightarrow (X[(\neg(\bigvee_i move\_to\_ball(i)))U(\bigvee_i kick\_ball(i))]}])$

Formula φ_1 enforces that, after one robot moves to the ball (which means the team does not have the ball in its

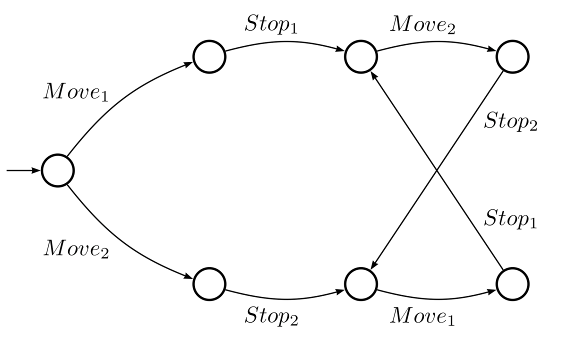

Fig. 8. Automaton representation of the controlled system, with the robot alternation implemented

possession), all the robots will not move to the ball until one of them shoots it (which means that the team lost the ball possession).

Spec $2, i$ is represented by formulas $\varphi_{2,i},\ i = 1, ..., n$, where

- $\varphi_{2,i} = ((\neg start\_receiving(i)) \wedge (G[(\bigvee_{j \neq i} start\_passing(j, i)) \Leftrightarrow (X start\_receiving(i))]))$

Formula $\varphi_{2,i}$ enforces that a robot's first action cannot be getting ready to receive a pass and that, only when one of its teammates chooses it as a receiver, it gets ready to receive the ball and it gets ready as soon as possible.

These formulas do not refer to uncontrollable events, so checking that an admissible supervisor is yield is immediate.

The controlled system was tested for 3 robots. The plant has 729 states, the supervisor obtained by φ_1 has 100 states (the great reduction in the number of states is due to the fact that the plant allows more than one robot to have the ball in its possession and it is φ_1 that disallows this kind of situation) and the supervisors obtained by $\varphi_{2,i},\ i = 1, 2, 3$ have 1458 states each. Next, we give two examples of output of the simulation. One should notice that when when one robot is chosen by a teammate to receive a pass, it gets ready to receive it immediately and that robots only go to the ball when it is not in the team's possession (i.e. when it is kicked) and only go one at a time. In Simulations 1 and 2 we emphasize these situations respectively. In Simulation 3 we show the uncontrolled behavior of the system. The lack of restrictions imposed for this system allows it to regularly evolve to a *deadlock* situation.

Simulation 1 - move_to_ball(3) - close_to_ball(3) - get_ball(3) - move_to_goal(3) - close_to_goal(3) - kick_ball(3) - move_to_ball(1) - close_to_ball(1) - get_ball(1) - blocked_path(1) - *start_passing(1,2)* - *start_receiving(2)* - *pass(1,2)* - blocked_path(2) - *start_passing(2,1)* - *start_receiving(1)* - *pass(2,1)* - move_to_goal(1) - blocked_path(1) - *start_passing(1,3)* - *start_receiving(3)* - *pass(1,3)* - move_to_goal(3) - close_to_goal(3) - kick_ball(3) - move_to_ball(3) - close_to_ball(3) - get_ball(3) - move_to_goal(3) - blocked_path(3) - *start_passing(3,1)* - *start_receiving(1)* - *pass(3,1)* - close_to_goal(1) - kick_ball(1) - move_to_ball(1) - close_to_ball(1) - get_ball(1) - blocked_path(1) - *start_passing(1,3)* - *start_receiving(3)* - *pass(1,3)* - blocked_path(3) - *start_passing(3,1)* - *start_receiving(1)* - *pass(3,1)* - move_to_goal(1) - blocked_path(1) - *start_passing(1,2)* - *start_receiving(2)* - *pass(1,2)* - close_to_goal(2) - kick_ball(2) - move_to_ball(3) - close_to_ball(3) - get_ball(3)

Simulation 2 - *move_to_ball(1)* - *close_to_ball(1)* - *get_ball(1)* - blocked_path(1) - start_passing(1,3) - start_receiving(3) - pass(1,3) - move_to_goal(3) - blocked_path(3) - start_passing(3,1) - start_receiving(1) - pass(3,1) - blocked_path(1) - start_passing(1,3) - start_receiving(3) - pass(1,3) - move_to_goal(3) - close_to_goal(3) - *kick_ball(3)* - *move_to_ball(2)* - *close_to_ball(2)* - *get_ball(2)* - blocked_path(2) - start_passing(2,3) - start_receiving(3) - pass(2,3) - blocked_path(3) - start_passing(3,2) - start_receiving(2) - pass(3,2) - close_to_goal(2) - *kick_ball(2)* - *move_to_ball(1)* - *close_to_ball(1)* - *get_ball(1)* - close_to_goal(1) - *kick_ball(1)* - *move_to_ball(2)* - *close_to_ball(2)* - *get_ball(2)* - close_to_goal(2) - kick_ball(2) - *move_to_ball(3)* - *close_to_ball(3)* - *get_ball(3)* - move_to_goal(3) - blocked_path(3) - start_passing(3,1) - start_receiving(1) - pass(3,1) - close_to_goal(1) - kick_ball(1)

Simulation 3 - start_receiving(3) - move_to_ball(1) - move_to_ball(2) - close_to_ball(2) - close_to_ball(1) - get_ball(1) - blocked_path(1) - start_passing(1,2) - get_ball(2) - blocked_path(2) - pass(1,3) - move_to_ball(1) - move_to_goal(3) - start_passing(2,1) - close_to_goal(3) - close_to_ball(1) - get_ball(1) - kick_ball(3) - close_to_goal(1) - kick_ball(1) - move_to_ball(3) - close_to_ball(3) - move_to_ball(1) - get_ball(3) - blocked_path(3) - start_passing(3,2) - close_to_ball(1) - get_ball(1) - move_to_goal(1) - blocked_path(1) - start_passing(1,2)

VII. Conclusion

In this work, we defined a method to perform supervisory control of Discrete Event Systems using Linear-Time Temporal Logic. We introduced all the necessary theory to understand how the method works and gave some examples of application. Analyzing the examples, one can conclude that, with this method, the specification of supervisors for systems with an arbitrary number of components that must coordinate themselves is almost straightforward: all the formulas are written for an arbitrary $n \in \mathbb{N}$. Unfortunately, this advantage is somewhat shadowed by the high complexity of the method: despite writing the formulas for an arbitrary number of components, when performing the simulations we witnessed the great increase of the number of states, both in the plant and in the supervisors, which only allows the application of the method for systems with a relatively small number of components.

There are several paths one can follow to improve the method we just presented. The most obvious one is to try to reduce its complexity. Another improvement is to increase the method's expressive power, for example by using CTL (a temporal logic that is incomparable with LTL) or CTL* (a temporal logic that contains both LTL and CTL) [6] as a way to specify the supervisors or by identifying each state of the DES model with a set of propositions that are satisfied in that state and build our LTL specification over those propositions, instead of building it over the DES' event set. One major advantage of this option is that it allows for more than one proposition to be satisfied at each state of the DES, unlike the method we presented, where only one is satisfied. One can also model the DES itself as a set of LTL formulas, as seen in [8], avoiding the construction of any automaton by hand (which can be very error-prone). Another option is to define a similar logic to LTL, but with its semantics defined over finite string, avoiding the need to use Büchi Automata. A final suggestion is to develop this theory in order to cover other aspects of Supervisory Control. For example, being concerned with marked languages and deal with blocking issues or introduce the notion of unobservable events [2].

REFERENCES

[1] Calin Belta, Antonio Bicchi, Magnus Egerstedt, Emilio Frazzoli, Eric Klavins and George J. Pappas, Symbolic Planning and Control of Robot Motion. In IEEE Robotics & Automation Magazine:61-70, March 2007

[2] Christos G. Cassandras and StŐphane Lafortune, *Introduction to Discrete Event Systems*, Kluwer Academic Publishers, 1999.

[3] E. Allen Emerson, Temporal and Modal Logic. In Handbook of theoretical computer science (vol. B): formal models and semantics, MIT Press:995 - 1072 1991

[4] Paul Gastin and Denis Oddoux, Fast LTL to Büchi Automata Translation, LIAFA, UniversitŐ Paris 7

[5] John E. Hopcroft, Motwani, Rajeev Ullman, Jeffrey D., *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*, Addison-Wesley, 2001

[6] Shengbing Jiang and Ratnesh Kumar, Supervisory Control Of Discrete Event Systems with CTL* Temporal Logic Specifications. In Proceeding of the 40th IEEE, Conference on Decision and Control:4122-4127, December 2001

[7] Marius Kloetzer and Calin Belta, A Fully Automated Framework for Control of Linear Systems From LTL Specifications. In Lecture Notes in Computer Science , J. Hespanha and A. Tiwari, Eds. Berlin, Germany: Springer-Verlag, vol 3927:333-347, 2006

[8] Jing-Yue Ling and Dan Ionescu, A Reachability Synthesis Procedure for Discrete Event Systems in a Temporal Logic Framework. In IEEE Transactions on Systems, Man, and Cybernetics, VOL. 24, No. 9:1397-1406, September 1994

[9] Peter J. G. Ramadge and W. Murray Wonham, The Control of Discrete Event Systems. In *Proceedings of the IEEE*, Vol. 77, No. 1:81-98, 1989.

[10] Paulo Tabuada and George J. Pappas, Linear Time Logic Control of Discrete-Time Linear Systems. In *IEEE Transactions on Automatic Control*, Vol. 51, No. 12:1862-1877, 2006.

[11] Pierre Wolper, Constructing Automata from Temporal Logic Formulas: A Tutorial. In *Lectures on Formal Methods in Performance Analysis*, Vol. 2090:261-277 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001