# Supervisory Control via Symbolic Model Checking

Andrey Gromyko, Marco Pistore
DIT, University of Trento
Via Sommarive 14, 38050 Povo, Trento, Italy
{gromyko,pistore}@dit.unitn.it

Paolo Traverso
SRA, ITC-IRST
Via Sommarive 18, 38050 Povo, Trento, Italy
traverso@itc.it

***Abstract*— We consider the supervisory control problem for discrete event systems. This problem refers to the automatic generation of the supervisor from the formal description of a plant and a specification of the desired behaviours. We provide a correct and complete symbolic algorithm for generic supervisor synthesis with temporal logic specifications. We build an algorithm that exploits efficient symbolic model checking techniques. Also, we consider two extreme cases of the problem: maximally permissive supervisor synthesis and directed control. Finally, we present a framework for efficient synthesis along with (controlled) plant simulation and verification capabilities.**

## INTRODUCTION

Discrete control systems are employed in many application domains, e.g., embedded systems, production cells in manufacturing systems. In these domains, control systems should often operate in a complex and rapidly changing environment, exhibiting a high degree of autonomy and meeting strong requirements with respect to safety, dependability, and responsiveness. Even in the cases where the applications are not safety-critical, high economical losses may be related to failures of controllers.

In the case of discrete event systems, the plant is described as a state-transition system that models the behaviours of the plant according to some possible events. The set of requirements (specifications) usually describes legal sequences of events. *Synthesis of supervisor* is the problem of translating a specification into a supervisor that is guaranteed to satisfy requirements.

The supervisory control problem is widely represented in the literature [13], [5], not to mention a lot more. The classical Ramadge-Wonham approach [13] has been extended to support efficient symbolic techniques, e.g. see [15]. However, it has been argued many times (e.g. [2], [14]) that specifying the requirements as an event-driven automaton, as requested in [13], [15], is not a trivial task for system designers. Indeed, it is hard to determine whether the prescriptive language of an automata really captures the desired control specification. To overcome this problem it was proposed to use temporal logic, which has long been recognized as an expressive and readable formalism for specifications in different contexts [11], [2], [7]. However, most papers on supervisory control with temporal logic specifications are dealing with theoretical aspects.

In this paper we present an efficient approach to supervisory control via symbolic techniques along with a framework for synthesis of supervisors. An algorithm takes as the input a plant (a state-transition system), requirements as CTL formulae (computational tree logic, see [9]), and produce a supervisor as the output. Proposed algorithms are implemented in the Model Based Planner (MBP) [3], a state-of-the-art system that has already proved its efficiency in the area of the planning for temporal logic specifications. We remark that latter shares several aspects with supervisory control sythesis. Apart from the synthesis, MBP allows one to simulate behaviours of both the uncontrolled plant and the controlled one. Since the framework is built on top of the NuSMV Symbolic Model Checker [6], we benefit also from the common language. Once a plant is implemented in SMV language, it can be efficiently verified using this model checker. Such a verification might be extremely useful to ensure the correctness of the complex plants' models.

The paper is structured as follows. Section I provides a formal definition of the general supervisory control problem. Section II describes two important subclasses of the latter, namely, maximally permissive and directed control. Section III presents a generic algorithm in detail, followed by a description of special cases for the subclasses. Section V provides some remarks on the implementation and experiments. Section VI discusses the related work. Finally, we conclude with a summary and some points for the future work.

## I. SUPERVISORY CONTROL

A *discrete event system* (DES) or a *plant* is a dynamic system that evolves in accordance with the abrupt occurrence, at possibly unknown irregular intervals, of physical events [13]. For example, a production line or any complex electronic system can be viewed as a such system. The set of events is partitioned into *uncontrollable* and *controllable* events. Intuitively, uncontrollable events are always enabled, while controllable events can be prevented from occurring by some external *supervisor (controller)* at any time. A supervisor follows the changes in a plant's state in order to issue *control masks* which keep a system's behaviour inside a desired *specification*.

### A. Plant, Supervisor, and Controlled plant

*Definition 1 (plant):* The *uncontrolled discrete event plant* $\mathcal{P}$ is modeled by a tuple

$$\mathcal{P} = \langle \mathcal{X}, \Sigma, \mathcal{AP}, \delta_P, x_0, \mathcal{L}_\mathcal{P} \rangle,$$

where:

- $\mathcal{X}$ is a finite set of *states*.
- $\Sigma = \Sigma^c \cup \Sigma^u$ is a finite set of *events* that is a disjoint union of $\Sigma^c$, the set of *controllable events*, and $\Sigma^u$, the set of *uncontrollable events*.

- $\mathcal{AP}$ is the finite set of *atomic proposition symbols*.
- $\delta_P : \mathcal{X} \times \Sigma \rightharpoonup \mathcal{X}$ is a partial *transition function* defined at each state of $\mathcal{X}$ for a subset of $\Sigma$.
- $x_0 \in \mathcal{X}$ is the *initial state* of $\mathcal{P}$.
- $\mathcal{L}_\mathcal{P} : \mathcal{X} \to 2^{\mathcal{AP}}$ is a *labelling function*.

Notice that $\mathcal{P}$ is called deterministic due to the deterministic behaviour of the transition function $\delta_P$ that is, for each state-event pair $(x, \sigma)$ with $x \in \mathcal{X}$ and $\sigma \in \Sigma$, $\delta_P(x, \sigma)$ is either not defined or produces a unique outcome $x' \in \mathcal{X}$. We write $!\delta_P(x, \sigma)$ if $\delta_P$ is defined for the pair $(x, \sigma)$ (the notation can be naturally extented to finite sequences $w = \sigma_1 \dots \sigma_n$). An event $\sigma$ is said to be *possible* in the state $x$ if $!\delta_P(x, \sigma)$. We require a plant to be *non-terminating*, that is at every state there is at least one possible event. Formally, a plant is *non-terminating* iff $\forall x \in \mathcal{X}\ \exists \sigma \in \Sigma : !\delta_P(x, \sigma)$.

*Definition 2 (supervisor):* A *supervisor* $\mathcal{S}$ for a plant $\mathcal{P}$ is modelled by a tuple:

$$\mathcal{S} = \langle \mathcal{Y}, y_0, \Gamma, \gamma, \delta_S \rangle,$$

where:

- $\mathcal{Y}$ is a set of *states*.
- $y_0 \in \mathcal{Y}$ is the *initial state* of $\mathcal{S}$.
- $\Gamma = 2^{\Sigma^c}$ is a set of control masks, essentially sets of forbidden controllable events.
- $\gamma : \mathcal{X} \times \mathcal{Y} \rightharpoonup \Gamma$ is a *control function* that given a pair of the plant's current state and the supervisor's current state returns the control mask, which enumerates currently forbidden controllable events.
- $\delta_S : \mathcal{X} \times \mathcal{Y} \times \mathcal{X} \rightharpoonup \mathcal{Y}$ is a partial *transition function*.

We denote by $\text{Ctrl}(x, \gamma) \subseteq \mathcal{X}$ the set of plant's states that can be reached from $x \in \mathcal{X}$ by all possible events $\sigma \in \Sigma \backslash \gamma$: $\text{Ctrl}(x, \gamma) = \{x' : \exists \sigma \in \Sigma \backslash \gamma\ .\ !\delta_P(x, \sigma) = x'\}$.

The controlled (supervised) plant $\mathcal{P}||\mathcal{S}$ is obtained by the strict synchronous composition of $\mathcal{P}$ and $\mathcal{S}$.

*Definition 3 (controlled plant):* A controlled plant is modeled by six tuple:

$$\mathcal{P}||\mathcal{S} = \langle \mathcal{Z}, \Sigma, \mathcal{AP}, \delta_{\mathcal{P}||\mathcal{S}}, z_0, \mathcal{L}_{\mathcal{P}||\mathcal{S}} \rangle,$$

where:

- $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$ is the set of states.
- $\Sigma, \mathcal{AP}$ are the same sets as given in $\mathcal{P}$.
- $\delta_{\mathcal{P}||\mathcal{S}} : \mathcal{Z} \times \Sigma \rightharpoonup \mathcal{Z}$ is the state transition function. Let $\sigma \in \Sigma$ and $(x, y) \in \mathcal{X} \times \mathcal{Y} = \mathcal{Z}$, then

$$\delta_{\mathcal{P}||\mathcal{S}}((x, y), \sigma) = \begin{cases} (\delta_P(x, \sigma), y') : !\delta_P(x, \sigma), \\ \qquad !\delta_S(x, y, \delta_P(x, \sigma)) = y' \\ \qquad \textit{and } \sigma \in \Sigma \backslash \gamma(x, y) \\ \textit{undefined}, \text{ otherwise.} \end{cases}$$

- $z_0 = (x_0, y_0) \in \mathcal{Z}$ is the *initial state* of $\mathcal{P}||\mathcal{S}$.
- $\mathcal{L}_{\mathcal{P}||\mathcal{S}} : \mathcal{Z} \to 2^{\mathcal{AP}}$ is a *labelling function* with $\mathcal{L}_{\mathcal{P}||\mathcal{S}}(z) = \mathcal{L}_{\mathcal{P}||\mathcal{S}}((x, y)) = \mathcal{L}_\mathcal{P}(x)$.

The transition function can be easily and naturally expanded to finite words in $\Sigma^*$. We say, that state $z = (x, y)$ is *reachable* if there exists finite sequence $\omega \in \Sigma^* : z = \delta_{\mathcal{P}||\mathcal{S}}(z_0, \omega)$. The supervisor $\mathcal{S}$ is said to be *non-blocking* for the plant $\mathcal{P}$ if for any reachable state $z\ \exists \sigma : !\delta_{\mathcal{P}||\mathcal{S}}(z, \sigma)$.

### B. Specification language: CTL

Formal specifications are expressed by CTL formulae. CTL allows temporal operators that define temporal conditions on a plant evolution. We assume that the set $\mathcal{AP}$ is defined for a plant $\mathcal{P}$. For all $a \in \mathcal{AP}$ and $x \in \mathcal{X}$, predicate $x \models_0 a$ holds iff $a \in \mathcal{L}_\mathcal{P}(x)$. If there is a need to express conditions on events occurences or sequences, one can easily extend a plant with a state variable corresponding to an occured event.

*Definition 4 (CTL):* The specification language CTL is defined by the grammar:

$$\begin{aligned} s &::= p \mid s \wedge s \mid s \vee s \mid \mathrm{AX}\, s \mid \mathrm{EX}\, s \\ &\qquad \mathrm{A}(s\, \mathrm{U}\, s) \mid \mathrm{E}(s\, \mathrm{U}\, s) \mid \mathrm{A}(s\, \mathrm{W}\, s) \mid \mathrm{E}(s\, \mathrm{W}\, s) \\ p &::= \top \mid \bot \mid a \mid \neg p \mid p \wedge p \end{aligned}$$

where $a \in \mathcal{AP}$.

CTL combines temporal operators and path quantifiers. "X", "U", and "W" are the "next time", "(strong) until", and "weak until" temporal operators, respectively. "A" and "E" are the universal and existential path quantifiers, where a path is an infinite sequence of states. They allow us to specify requirements that take into account nondeterminism.

Intuitively, the formula $\mathrm{AX}\, s$ means that $s$ holds in every immediate successor of the current state, while the formula $\mathrm{EX}\, s$ means that $s$ holds in some immediate successor. The formula $\mathrm{A}(s_1\, \mathrm{U}\, s_2)$ means that for every path there exists an initial prefix of the path, such that $s_2$ holds at the last state of the prefix and $s_1$ holds at all the other states along the prefix. The formula $\mathrm{E}(s_1\, \mathrm{U}\, s_2)$ expresses the same condition, but only on some of the paths. The formulae $\mathrm{A}(s_1\, \mathrm{W}\, s_2)$ and $\mathrm{E}(s_1\, \mathrm{W}\, s_2)$ are similar to $\mathrm{A}(s_1\, \mathrm{U}\, s_2)$ and $\mathrm{E}(s_1\, \mathrm{U}\, s_2)$, but allow for paths where $s_1$ holds in all the states and $s_2$ never holds. Formulae $\mathrm{AF}\, s$ and $\mathrm{EF}\, s$ (where the temporal operator "F" stands for "future" or "eventually") are abbreviations of $\mathrm{A}(\top\, \mathrm{U}\, s)$ and $\mathrm{E}(\top\, \mathrm{U}\, s)$, respectively. $\mathrm{AG}\, s$ and $\mathrm{EG}\, s$ (where "G" stands for "globally" or "always") are abbreviations of $\mathrm{A}(s\, \mathrm{W}\, \bot)$ and $\mathrm{E}(s\, \mathrm{W}\, \bot)$, respectively.

Notice, that even if negation $\neg$ is allowed only in front of atomic propositions, it is easy to define $\neg s$ for a generic CTL formula $s$, by pushing down the negations. For instance, $\neg \mathrm{AX}\, s \equiv \mathrm{EX}\, \neg s$ and $\neg \mathrm{A}(s_1\, \mathrm{W}\, s_2) \equiv \mathrm{E}(\neg s_2\, \mathrm{U}(\neg s_1 \wedge \neg s_2))$.

Specifications (properties) as CTL formulae allow us to specify different classes of requirements on controllers. Let us consider first some examples of *reachability properties*. $\mathrm{AF}\, s$ (reach $s$) states that a condition should be guaranteed to be reached by a controlled plant, in spite of nondeterminism. $\mathrm{EF}\, s$ (try to reach $s$) states that a condition might possibly be reached, i.e., there exists at least one evolution that satisfies the specification. A reasonable reachability requirement that is stronger than $\mathrm{EF}\, s$ is $\mathrm{A}(\mathrm{EF}\, s\, \mathrm{W}\, s)$: it allows for those evaluation loops that have always a possibility of terminating, and when they do, the specification $s$ is guaranteed to be satisfied.

We can distinguish also among different kinds of *maintainability properties*, e.g., $\mathrm{AG}\, s$ (maintain $s$), $\mathrm{AG}\, \neg s$ (avoid $s$), $\mathrm{EG}\, s$ (try to maintain $s$), and $\mathrm{EG}\, \neg s$ (try to avoid $s$).

The "until" operators $\mathrm{A}(s_1 \,\mathrm{U}\, s_2)$ and $\mathrm{E}(s_1 \,\mathrm{U}\, s_2)$ can be used to express the reachability properties $s_2$ with the additional requirement that property $s_1$ must be maintained until the desired condition is reached.

We can also compose reachability and maintainability properties in arbitrary ways. For instance, $\mathrm{AF\,AG}\, s$ states that a supervisor should guarantee that all evolutions eventually reach a set of states where $s$ can be maintained. The weaker specification $\mathrm{EF\,AG}\, s$ requires that there exists a possibility to reach a set of states where $s$ can be maintained. As a further example, the specification $\mathrm{AG\,EF}\, s$ intuitively means "maintain the possibility of reaching $s$".

Notice that in all examples above, the ability of composing formulae with universal and existential path quantifiers is essential. Logics that do not provide this ability, like LTL [9], cannot express these kinds of specifications[1].

We remark that a controlled plant is a Kripke structure. Thus, we can use the standard semantics for CTL formulae over Kripke structures [9]. We say that supervisor $\mathcal{S}$ satisfies specification $f$ for a plant $\mathcal{P}$, or $\mathcal{P}||\mathcal{S} \models f$ if $f$ is true in all initial states of the Kripke structure $\mathcal{P}||\mathcal{S}$.

*Definition 5:* **Supervisory Control Problem** $(\mathcal{P}, f)$**:** For a given non-terminating plant $\mathcal{P}$ and specification $f$, find a non-blocking supervisor $\mathcal{S}$ such that $\mathcal{P}||\mathcal{S} \models f$.

## II. Degree of Control

There exist two extreme cases of the supervisory control problem. Namely, one can be interested in a maximally permissive control, while the other might want to have directed control over a system. Intuitively, maximally permissive supervisor should allow the widest possible set of controllable events, which guarantees a desired behaviour of a system. On the contrary, a director is intented to allow at most one controllable event at the time.

Below we provide a formal definition of a maximally permissive supervisor and a director.

### A. Maximally permissive supervisor

Let us denote the language generated by the controlled plant as $\mathcal{L}(\mathcal{P}||\mathcal{S}) \subseteq \Sigma^\omega$. That is an (infinite) events sequence $\forall s = \sigma_1\sigma_2\ldots,\ s \in \mathcal{L}(\mathcal{P}||\mathcal{S}) \Leftrightarrow \forall k > 0,\ x_k = \delta_{\mathcal{P}||\mathcal{S}}((x_0, y_0), s^k)$ is defined.

We say that for a plant $\mathcal{P}$ a supervisor $maxS$, such that $\mathcal{P}||maxS \models f$, is maximally permissive with respect to a specification $f$, if for any supervisor $\mathcal{S} : \mathcal{P}||\mathcal{S} \models f \Rightarrow \mathcal{L}(\mathcal{P}||\mathcal{S}) \subseteq \mathcal{L}(\mathcal{P}||maxS)$. It means that maximally permissive supervisor allows for all possible system behaviours which do not violate a specification.

From this formal description it follows that if there exists a maximally permissive supervisor for a problem $(\mathcal{P}, f)$, then it is unique.

[1]In general, CTL and LTL have incomparable expressive power [9]. We focus on CTL since it provides the ability of expressing specifications that take into account nondeterminism.

### B. Director

As opposed to the maximally permissive control, there is an interest [10] for the directed control of some systems. Formally, a *director* is a special case of supervisor with the following property: $\forall x \in \mathcal{X}, y \in \mathcal{Y}$ if $\gamma(x, y)$ is defined, then $|\Sigma^c| - |\gamma(x, y)| \leq 1$. That is, a director allows at most one controllable event in every state. Notice, that in general for a supervisory control problem $(\mathcal{P}, f)$ there may exist several different valid directors, if any.

## III. Symbolic synthesis algorithm

The synthesis algorithm takes in input a plant and a specification, and constructs a supervisor satisfying the specification for as many initial states as possible. To check whether a supervisor exists for a given initial state $x_0$ it is sufficient to check whether the control function is defined at $(x_0, y_0)$.

The outline of the symbolic synthesis algorithm is the following:

**function** *symbolicSupervisor*$(\mathcal{P}, f)$ : *Supervisor*
  *aut* := *buildCtrlAutomaton*$(f)$
  *assoc* := *buildAssoc*$(aut, \mathcal{P})$
  *supervisor* := *extractSupervisor*$(aut, assoc)$
  **return** *supervisor*

In the first step, *buildCtrlAutomaton* constructs the control automaton, a specialized nondeterministic tree automaton, which captures all possible variants to satisfy a given specification $f$. An automaton built in this step is used to control the symbolic search performed in the following *buildAssoc* function call. Thus, *buildAssoc* exploits the control automaton to guide the symbolic exploration of a plant. This function associates a set of states in the plant to each state in the control automaton. Intuitively, these are the states for which a supervisor exists from the given control state. Finally, *extractSupervisor* constructs a supervisor by exploiting the information on the states associated to the control automaton states.

This section is structured according to the algorithm steps. First, we provide details on the control automata construction procedure. Then we thoroughly describe the way of associating the plant states to the control automaton states. Finally, we remark on the supervisor extraction procedure.

### A. Control automata construction

The nodes of the automaton ("control states" or contexts) correspond to the current (active) subspecifications that need to be resolved by the algorithm. These nodes constitute the set of states of the supervisor, which is being synthesized. The transitions correspond to possible evolutions of control states during the search.

Consider, for example, the specification $\mathrm{AG}\, p \wedge \mathrm{EF}\, q$: the supervisor must guarantee that $p$ is always maintained and that there is a possibility to reach $q$. In this case, the algorithm has to deal with two cases: (*i*) $q$ has still to be reached and the algorithm must both preserve the chance to reach $q$, thus satisfying $\mathrm{EF}\, q$, and guarantee that $p$ is maintained,

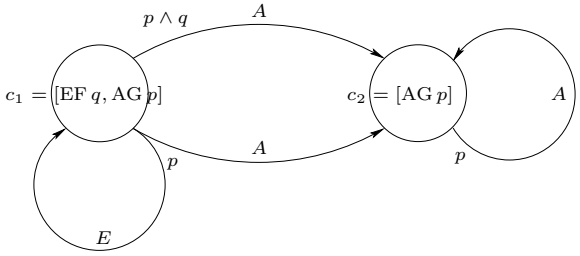


Fig. 1. The control automaton for $\mathrm{AG}\, p \wedge \mathrm{EF}\, q$

thus satisfying $\mathrm{AG}\, p$; (*ii*) $q$ has been already reached and the algorithm must guarantee maintenance of $p$ ($\mathrm{AG}\, p$). Function *buildCtrlAutomaton* constructs the automaton, presented in Figure 1, which represents these two cases. Let us consider the state $c_1 = [\mathrm{EF}\, q, \mathrm{AF}\, p]$ of the automaton. This control state corresponds to the situation where synthesis algorithm has still to satisfy both $\mathrm{EF}\, q$ and $\mathrm{AG}\, p$ requirements. From the state $c_1$ we have two possible transitions, corresponding to the two cases described above. The transition labeled with $p \wedge q$ corresponds to the case when $q$ is reached, and thus, leads to the state $c_2 = [\mathrm{AG}\, p]$. The transition is also marked with $A$ to guarantee maintainability of $p$ (universal path quantifier). In the control state $c_2$ property $p$ must be maintained forever (self-transition marked with $p$). The second transition from $c_1$ corresponds to the case when only $p$ holds, but $q$ was not reached yet. The transition forks in two states to denote that the conjunction of two requirements must be satisfied. The supervisor has to guarantee both that in all the next states $p$ is maintained (arc to $c_2$ marked with $A$), and that there is at least one next state where $\mathrm{EF}\, q$ is satisfied (self-transition marked with $E$).

In the above example, the two control states $c_1$ and $c_2$ induce different kinds of symbolic operations due to the different semantics of associated requirements. In the case of $c_2$, $\mathrm{AG}\, p$ is a maintanability requirement. The algorithm starts from states where $p$ holds and eliminates iteratively those states that lead to the next states where $\mathrm{AG}\, p$ does not hold. Technically, a greatest fixpoint is performed[2]. In the case of $c_1$, in order to deal with the reachability requirement $\mathrm{EF}\, q$, the algorithm starts from states where $q$ holds and adds iteratively the states where $\mathrm{EF}\, q$ is achieved for some next states. Technically, a least fixpoint is performed. We remark, that the different computational treatment is caused by differences in the semantics of (strong) until and weak until CTL operators. We call sets of control states of the kind of $c_2$ and of the kind of $c_1$ *green* and *red blocks*, respectively. These blocks define *acceptance conditions* on the infinite runs of the supervisor similar to those defined in the field of automata theory.

Let us introduce a formal definition of the control automaton.

*Definition 6 (control automaton):* A *control automaton* is a tuple $A_{ctrl} = \langle C, c_0, T, R \rangle$, where:

- $C$ is the set of contexts, i.e. (control) states of the automaton;
- $c_0 \in C$ is the initial context;
- $T : C \rightarrow 2^{\mathcal{P}rop(\mathcal{B}) \times C \times 2^C}$ is the transition function, where $\mathcal{P}rop(\mathcal{B})$ is any propositional formula constructed from the basic propositions $b \in \mathcal{B}$;
- $R = \{B_1, \ldots, B_n\}$, with $B_i \subseteq C$, is the set of red blocks of the automaton.

Each triple $(P, A, Es) \in T(c)$ represents one of the possible evolutions from the control state $c$. $P$ constrains properties of the states where a particular evolution is applicable. $A$ is the (sub)specification that must hold for all the next states. $Es$ is a set of requirements to be held for some next states.

For the example in Figure 1, we have: $C = \{c_1, c_2\}$, $T(c_1) = \{(p \wedge q, c_2, \emptyset), (p, c_2, \{c_1\})\}$, $T(c_2) = \{(p, c_2, \emptyset)\}$, the initial context is $c_1$, the set of red blocks $R = \{B_1\}$ with $B_1 = \{c_1\}$.

We now formalize how the control automaton is constructed from a specification. One of the key basic functions in the construction of the control automata is the function *progr*, which associates to each specification $f$ the conditions that the specification defines on the current state and on the next states to be reached, according to the CTL semantics:

- $\mathit{progr}(\top) = \top$, $\mathit{progr}(\bot) = \bot$, $\mathit{progr}(b) = b$, $\mathit{progr}(\neg b) = \neg b$;
- $\mathit{progr}(f_1 \wedge f_2) = \mathit{progr}(f_1) \wedge \mathit{progr}(f_2)$, $\mathit{progr}(f_1 \vee f_2) = \mathit{progr}(f_1) \vee \mathit{progr}(f_2)$;
- $\mathit{progr}(\mathrm{AX}\, f) = \mathrm{AX}\, f$, $\mathit{progr}(\mathrm{EX}\, f) = \mathrm{EX}\, f$;
- let $\circ$ be either the (strong) until operator U or the weak until operator W, then: $\mathit{progr}(\mathrm{A}(f_1 \circ f_2)) = (\mathit{progr}(f_1) \wedge \mathrm{AX}\, \mathrm{A}(f_1 \circ f_2)) \vee \mathit{progr}(f_2)$, and $\mathit{progr}(\mathrm{E}(f_1 \circ f_2)) = (\mathit{progr}(f_1) \wedge \mathrm{EX}\, \mathrm{E}(f_1 \circ f_2)) \vee \mathit{progr}(f_2)$.

The formula $\mathit{progr}(f)$ can be rewritten in a disjunctive normal form. Each disjunct consists of the conjunction of three kinds of formulae: the propositional ones (a condition for current states), those of the form $\mathrm{AX}\, \varphi$ (formulae that must hold in all next states), and those of the form $\mathrm{EX}\, \psi$ (formulae that must hold in some of next states):

$$\mathit{progr}(f) = \bigvee_{i \in I} \Big( \bigwedge_{\varphi \in A_i} \mathrm{AX}\, \varphi \wedge \bigwedge_{\psi \in E_i} \mathrm{EX}\, \psi \Big)$$

where $\varphi \in A_i$ ($\psi \in E_i$) if $\mathrm{AX}\, \varphi$ ($\mathrm{EX}\, \psi$) belongs to the *i*-th disjunct of $\mathit{progr}(f)$. We have $|I|$ different disjuncts that correspond to alternative evolutions of a plant, i.e., to alternative supervisors we can search for. In the following, we represent $\mathit{progr}(f)$ as a set of triples, namely:

$$\mathit{progr}(f) = \{(P_i, AX_i, EX_i) \mid i \in I\}.$$

The result returned by *progr* is used to guide the construction of the control automaton. Since the component EX of *i*-th disjunct may contain any number of subformulae, there exist different ways to distribute them between the next states. Let $EX_1, \ldots, EX_k$ be an arbitrary partition of $EX$, then $\forall j = 1..k$ there must be some next state where

[2]For details on how the set of states satisfying CTL formula can be characterized as fixpoint computations, see [7].

subformulae $EX_j$ hold. Subformulae in $AX$ must hold in all the next states.

In order to formalize the construction of the red and green blocks, we need to distinguish three kinds of formulae: *(strong) until*, *weak until*, and *transient* ($\mathrm{AX}\_$, $\mathrm{EX}\_$, $\_\vee\_$, $\_\wedge\_$) requirements. Transient requirements are "resolved" in one step: AX and EX prefixes are being removed during the progress, and the logical operators $\vee$ and $\wedge$ are being split up. Weak until requirements are allowed to hold forever. On the contrary, strong until requirements must be resolved for a supervisor to be valid. To guarantee that the algorithm eventually resolves strong until requirements, we structure a context in the automaton as an ordered list of subformulae, with the priority given to strong until requirements. Among the latter we give priority to requirements that are active since more steps, i.e. the longer strong until formula is active and unresolved, the "more urgent" it becomes. The first formula ($\mathit{head}(c)$) in a context is the most urgent one.

A valid supervisor should not allow for executions where a strong until requirement becomes active and then never resolved. In order to capture this, the construction procedure generates red blocks as a sets of contexts that share the same most urgent requirement. Thus, a valid supervisor must guarantee that for all executions, if $B_i$ is entered ($\mathit{head}(c)$ has became active), then the red block is eventually left (context $c \notin B_i$ is reached).

*Definition 7 (control automaton construction):* The control automaton $A_{ctrl} = \mathit{buildCtrlAutomaton}(f)$ for requiremets $f$ is built according to the following rules:

- $c_0 = [f] \in C$;
- If $c = [f_1, \ldots, f_n] \in C$ then for each $(P, AX, EX) \in \mathit{progr}(f_1 \wedge \cdots \wedge f_2)$ and for each partition $\{EX_1, \ldots, EX_k\}$ of $EX$:

  $$(P, \mathit{order}(AX), \{\mathit{order}(AX \cup EX_j) : j = 1..k\}) \in T(c).$$

  Moreover, $\mathit{order}(AX) \in C$ and $\mathit{order}(AX \cup EX_j) \in C$ for each $j = 1..k$.
- For each strong until subformula $f'$ of $f$ let $B = \{c \in C : \mathit{head}(c) = f'\}$; if $B \neq \emptyset$, then $B \in R$.

We remark, that contexts are essentially sets of subformulae.

### B. Associating plant states to contexts

Once the control automaton for a specification $f$ is built, the synthesis algorithm proceeds by associating to each state in the automaton a set of states in the uncontrolled plant. The association is built by the function *buildAssoc*:

```
1  function buildAssoc(aut, P) : Assoc
2    foreach c ∈ aut.C do assoc[c] := P.X
3    greenBlock := {c ∈ C : ∀B ∈ aut.R . c ∉ B}
4    blocks := aut.R ∪ {greenBlock}
5    while (∃B ∈ blocks . canRefine(B)) do
6      if B ∈ aut.R then
7        foreach c ∈ B do assoc[c] := ∅
8      while (∃c ∈ B . canUpdate(c)) do
9        assoc[c] := updateCtxt(aut, assoc, c)
10   return assoc
```

The algorithm starts with an optimistic association, which assigns all the states $\mathcal{X}$ of a plant to each control state (line 2). This association is then iteratively refined in the following manner. At every iteration of the loop (lines 5-9), a block of contexts is choosen, and corresponding associations are updated. Those states are removed from the association, from which the algorithm discovers that the context requirements are not satisfiable. The algorithm terminates when a fixpoint is reached, that is, whenever no further refinement of the association is possible (in this case function $\mathit{canRefine}(B)$ in line 5 evaluates to false for each $B \in \mathit{blocks}$ and the guard of the correspondent **while** fails). The chosen block may be either one of the red blocks or from the block of states that are not in any red block, i.e., from the green block of an automaton.

In the case of the green block, the refinement step must guarantee only that all the states associated to the contexts are "safe", that is they never lead to contexts where the specification cannot be satisfied anymore. This refinement (lines 8-9) is obtained by choosing a context in the green block and then refreshing the associated set of states with the function *updateCtxt*. Once the fixpoint is reached and all the refresh steps on the states in $B$ do not change the association (*canUpdate* evaluates to false), the loop is left, and another block is chosen.

A red block consists of all contexts correspondent to a given most urgent strong until subspecification $f$. In this case, the refinement guarantees not only that the states in the association are "safe", but also that subspecification $f$ is eventually resolved, that is the correspondent red block is eventually left. For this purpose, the sets of states associated to red block contexts are initially emptied (lines 6-7). Then each context is updated iteratively (lines 8-9). In this way, a least fixpoint is computed for the states associated to the red block.

The core step of *buildAssoc* is the function $\mathit{updateCtxt}(\mathit{aut}, \mathit{assoc}, c)$. It takes in input a control automaton *aut*, a current association of states *assoc*, a context $c \in C$, and returns a new set of states to be associated to $c$.

$$\begin{aligned}\mathit{updateCtxt}(\mathit{aut}, \mathit{assoc}, c) \triangleq \{x \in \mathcal{X} :\ & \\ & \exists \gamma \in \Gamma, \exists (P, A, Es) \in T(c) \\ & x \in \mathit{statesOf}(P) \wedge \\ & (x, \gamma) \in \mathit{strongPreImage}(\mathit{assoc}[A]) \wedge \\ & (x, \gamma) \in \mathit{multiWeakPreImage}(\{\mathit{assoc}[E] : E \in Es\})\}.\end{aligned}$$

For a state to be associated to a context it is sufficient that a control mask $\gamma \in \Gamma$ exists such that next states satisfy the transition conditions of the automaton. Let us consider an element $(P, A, Es) \in T(c)$ and a control mask $\gamma$. Formula $P$ describes conditions on the current states: only states, which satisfy property $P$ are valid (guaranteed by the condition $x \in \mathit{statesOf}(P)$). $A$ is a context that should hold in all the next states. In order to satisfy this constraint, the function *strongPreImage* is exploited on the set $\mathit{assoc}[A]$ of states

associated to context $A$. Function *strongPreImage*$(X)$ returns the state-mask pairs that guarantee to reach states in $X$:

$$\textit{strongPreImage}(X) \triangleq \{(x,\gamma) : \text{Ctrl}(x,\gamma) \subseteq X\}.$$

Set $Es$ contains contexts that must be reached for some next states. To satisfy this constraint, function *multiWeakPreImage* is called on the set $\{\textit{assoc}[E] : E \in Es\}$, the elements of which are sets of states associated to contexts in $Es$.

$$\textit{multiWeakPreImage}(Xs) \triangleq \{(x,\gamma) : \exists i : Xs \mapsto \text{Ctrl}(x,\gamma)\ .\ \forall X \in Xs \Rightarrow i(X) \in X\}.$$

This function can be seen as a generalization of the function *weakPreImage*$(X)$, which computes state-mask pairs that may lead to a state in $X$: *weakPreImage*$(X) = \{(x,\gamma) : \text{Ctrl}(x,\gamma) \cap X \neq \emptyset\}$. In the function *multiWeakPreImage* an injective map is required to exist from *Xs* to the next states reachable under a control mask $\gamma$. This map guarantees that there is at least one next state in every set of states in *Xs*: $\forall X \in Xs \Rightarrow \text{Ctrl}(x,\gamma) \cap X \neq \emptyset$.

The function *updateCtxt* is the performance critical step of the algorithm for large plants. Indeed, it is the step where the plant is explored to compute preimages of sets of states. BDD-based symbolic techniques [4] are exploited in this step to obtain a compact representation of the sets of states associated to contexts, and to allow for an efficient exploration of the plant. Symbolic techniques allow for representing sets of states in a plant and relations on these sets as logic formulae over the atomic propositions assigned to the states. In particular, the transition relation of a plant can be represented symbolically, and this is intensively used in functions like *strongPreImage* and *weakPreImage*.

### C. Extracting the supervisor

Once the association *assoc* of the states to contexts is built for an automaton *aut*, a supervisor can be obtained. For lack of space, we provide only hints on how the function *extractSupervisor* works.

In particular, the set of control states of *aut* coincides with the set of supervisor states. The information necessary to define function $\gamma$ and transition function $\delta_S$ is implicitly computed in *buildAssoc*. To be more precise, functions *updateCtxt* and *multiWeakPreImage* determine, respectively, the control mask $\gamma(x,y)$ to be applied at a given plant state $x$ and a control (context) state $y$, and the next control state $\delta_S(x,y,x')$ for every possible next state $x'$. Thus, to extract a supervisor from a given association there is a need only to collect this information and explicitly define correspondent functions.

The algorithm always terminates, it is correct and complete.

*Theorem 1:* Let $\mathcal{P}$ be a plant, and $f$ be a specification for $\mathcal{P}$. Then *symbolicSupervisor*$(\mathcal{P}, f)$ terminates returning a supervisor candidate $\mathcal{S}$. If $\mathcal{S}.\gamma(x_0,y_0)$ is not defined, then $\nexists \mathcal{S}$ such that $\mathcal{P}||\mathcal{S} \models f$, otherwise computed supervisor $\mathcal{S}$ guarantees that $\mathcal{P}||\mathcal{S} \models f$.

## IV. Extreme cases

In the previous section we presented the algorithm solving a generic supervisory control problem. In order to synthesize a director, one needs to interfere only at the last step of the algorithm. Namely, the extraction of the supervisor should be performed taking into account that at most one controllable event should be allowed at every state.

More challenging is the case of maximally permissive supervisor synthesis. The first source of problems for such synthesis is the semantics of the "(strong) until" temporal operator of CTL. It is the source of control automaton red blocks and only operator which has "lasting finite" semantics. That is, it requires an unspecified, but still finite, number of steps to resolve some desired property (see formal definition in [9]). Technically, these blocks are resolved by a least fixpoint computation, which in fact chooses a minimal path to satisfy the requirement despite other (longer) possibilities.

The second obstacle for the maximally permissive supervisor synthesis is the operator $\vee$. Indeed, considering an illustrative example of the specification formula $f = AXp \vee AXq$, one can immediately end up in the following situation.

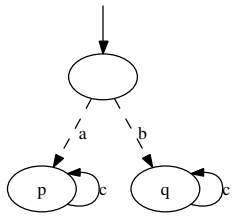


Fig. 2. The plant counter-example for $f = \mathrm{AX}\,p \vee \mathrm{AX}\,q$

There exist two different supervisors for the plant in Figure 2 which satisfy $f$ (the one denying the controllable event "a", and the second denying the controllable event "b"), but none of them is maximally permissive. A similar problem arises for $\mathrm{A}(\varphi\,\mathrm{W}\,\psi)$ since immediately we have $\textit{progr}(\mathrm{A}(\varphi\,\mathrm{W}\,\psi)) = \psi \vee (\varphi \wedge \mathrm{AX}\,\mathrm{A}(\varphi\,\mathrm{W}\,\psi))$.

Taking these into account, we can now formalize the needed restriction of the specification language.

### A. CTL fragment for maximally permissive supervisors

Let us present a fragment of CTL, which allow for a maximally permissive supervisor synthesis.

*Definition 8:* Let $a \in \mathcal{AP}$ a *specification* is a formula $f$, such that:

$$f ::= p \mid p \vee f \mid f \wedge f \mid \mathrm{AX}\,f \mid \mathrm{A}(f\,\mathrm{W}\,p)$$
$$p ::= \top \mid \bot \mid a \mid \neg p \mid p \wedge p \mid p \vee p$$

Intuitively, one can specify different safety specifications and/or production plans (e.g., desired assembly sequences) with this fragment of CTL. That is, $\mathrm{AX}\,p$ stands for "one (or next) step safety", when a supervisor should guarantee the property $p$ to hold in all the possible next states of a system. $\mathrm{A}(p\,\mathrm{W}\,q)$ is "to maintain (possibly infinite) property $p$ until

a system will reach a state, where $q$ holds". $\mathrm{AG}\, p$ abbreviates the special case $\mathrm{A}(p \,\mathrm{W}\, \bot)$ of the previous formula, that is when some property $p$ should be held forever.

We remark that the chosen fragment is the part of ACTL (universal fragment of CTL) and it is a proper subset of the common fragment of CTL and LTL [12]. Another observation is that the selected CTL fragment can be extended with existential path formulae $\mathrm{EX}\, f$ and $\mathrm{E}(f \,\mathrm{W}\, p)$ without any problem. However, there is no evidence for their necessity in the case of the maximally permissive supervisor synthesis.

## V. Implementation and experiments

The presented synthesis algorithm was implemented inside MBP [3]. MBP is implemented in C and uses BDD-based symbolic techniques to tackle the state-space explosion problem. We also implemented some tests, and results show that MBP efficiently solves both the classical maximally permissive controller synthesis problem [16] and the directed control problem [10].

For lack of space, here we present and discuss only one classical test case, namely, extended Transfer Line [17], [16]. We remind that a simple Transfer Line consists of two machines M1 and M2, followed by the special machine called test unit TU. There are also two buffers B1 and B2, the first between two machines, and the second between M2 and TU. Every machine has two states: idle and work. The buffers B1 and B2 have a capacity 3 and 1, correspondingly. In order to increase complexity of the problem, the simple Transfer Line example may be extended in two directions. One possibility is to allow all components to handle $M$ workpieces, see Figure 3 (controlled events are represented with dashed lines). The second is to connect $L$ Transfer

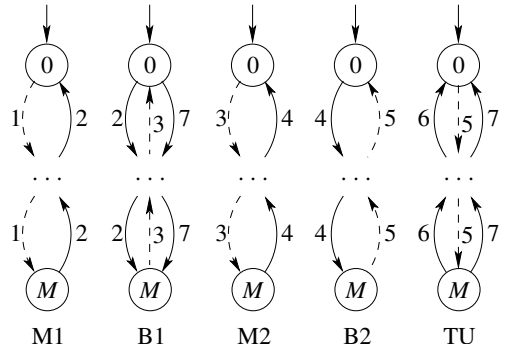


Fig. 3. Automaton model of the $M$-extended Transfer Line

Lines in a sequence with additional buffers between each pair of $M$-extended Tranfer Lines. Thus, we obtain an $(L, M)$-extended Transfer Line with $N = 5L + (L-1)$ components, since every Transfer Line has 5 components, and we need $L-1$ buffers to connect $L$ lines. Finally, the total number of states in the $(L, M)$-extended Transfer Line is $|states| = (M+1)^N$.

The tests were performed on the laptop equipped with Pentium-M 1.6GHz processor, 512MB RAM, and running GNU/Linux-2.6.13 operating system. Table I reports times required MBP to find the maximally permissive supervisor for some cases. The time limit was set to 30 minutes (1800 seconds). One may notice that algorithm suffers more from the number of controllable events $|\Sigma^c|$ than from the total number of states $|states|$. Indeed, the number of controllable events directly affects the number of control masks, or different control possibilities, which should be evaluated by the algorithm in the function *updateCtxt*. This suggests the first improvement of the algorithm: to provide more careful way of the preimages computation with respect to applicable control masks. It might be also adapted to particular extreme cases.

TABLE I
Results of experiments, $(L, M)$-extended Transfer Line

| $(L, M)$ | $\lvert states\rvert$ | $\lvert\Sigma^c\rvert$ | Time, sec |
|---|---|---|---|
| (1, 10) | $1.61 \times 10^5$ | 3 | 0.68 |
| (1, 20) | $4.08 \times 10^6$ | 3 | 1.98 |
| (1, 30) | $2.86 \times 10^7$ | 3 | 5.45 |
| (1, 45) | $2.06 \times 10^8$ | 3 | 43.65 |
| (1, 80) | $3.48 \times 10^9$ | 3 | 146.85 |
| (1, 100) | $1.05 \times 10^{10}$ | 3 | 299.76 |
| (1, 160) | $1.08 \times 10^{11}$ | 3 | 880.35 |
| (2, 2) | $1.77 \times 10^5$ | 6 | 2.11 |
| (2, 3) | $4.19 \times 10^6$ | 6 | 8.85 |
| (2, 4) | $4.88 \times 10^7$ | 6 | 34.07 |
| (2, 5) | $3.63 \times 10^8$ | 6 | 307.77 |
| (2, 6) | $1.98 \times 10^9$ | 6 | 1689.32 |
| (3, 1) | $1.31 \times 10^5$ | 9 | 3.78 |
| (3, 2) | $1.29 \times 10^8$ | 9 | 493.60 |
| (4, 1) | $8.39 \times 10^6$ | 12 | 265.82 |

To the best of our knowledge, the most relevant competitors to our tool are CTCT [16] and Supremica [1], both available online. We do not provide parallel evaluations, since both of them use explicit state-space representation, and thus, are not able to deal with majority of presented test cases. For example, in the same test environment, for (2,2)-extended TL, Supremica first reports "out of memory" error and recommends to increase Java virtual machine (JVM) memory heap. After assigning the minimum of 256MB of memory to JVM, Supremica manages to solve the problem in 883 seconds (MBP requires 2 sec, and <10MB).

We remark, that in the literature one can find references to the symbolic variants of both (STCT [17], and extented Supremica [15]). However, they are not publicly available. According to the private correspondence of the authors and Prof. Wonham, STCT is no longer supported. And we are looking forward to compare MBP with the symbolic implementation of Supremica in the equal test environment.

The implementation of MBP along with some examples is publicly available at http://sra.itc.it/tools/mbp/. In addition to the synthesis, MBP allows one to simulate a plant behaviours both standalone and with a synthesized control applied. The proposed approach benefits from the use of the SMV language. The input plant can be verified by the NuSMV model checker as a whole system or on a module basis. Such verified module encodings can be re-used to model complex flexible plants.

## VI. Discussion

The proposed framework allows for the supervisor synthesis in a widely-used setting. For example, it can accept as an input the classical Ramadge-Wonham problem encodings used in CTCT and Supremica, that is when both a plant and a specification are directly modeled as automata. However, we agree with many authors (e.g. [14], [2]) that for system designers it is quite hard (i) to model a specification as an automata, and (ii) to read such specifications. Moreover, we remark that such plant and specification implementations may lead to ambiguities and misinterpretations.

Indeed, let us consider the Transfer Line example. The requirement stated in the natural language is "to prevent buffers from underflow and overflow". In the Ramadge-Wonham framework the system is modelled as follows: machines belong to the plant model, while buffers (still physical parts of the plant) are becoming specifications. First, the method of the partitioning of generic plant components to "plant" and "specification" is not so clear. Second, the good (marked) states of both, which are needed for the exploited reachability analysis algorithms, may lead to the misinterpretation of a model. For the same example, since marked states for both machines and buffers is "0", one may understand that a good state for a machine is "idle" and for a buffer to be empty (compare to the intended specification). From the other side, the specification in a natural language can be clarified: "whenever the buffer is empty(full), no take(put) event should occur". The latter can be easily formalized in the CTL formula:
$\mathrm{AG}\left((B\_empty \rightarrow event \neq take) \wedge (B\_full \rightarrow event \neq put)\right)$,
which remains readable without any ambiguity.

Thus, we believe that the use of logic formalisms, e.g. CTL, for specifying requirements is more natural and easier for system designers. Such an approach allows one to separate the process of a plant modeling from setting the requirements.

## VII. Conclusions and Further work

We propose the symbolic algorithm for supervisory control synthesis with CTL specification. Also, we consider to important subclasses of the problem, namely, the maximally permissive and directed control cases. From the practical side, we present the efficient implementation of the synthesis algorithm based on state-of-the-art techniques used in the symbolic model checker NuSMV. We provide the preliminary report and discussion of the performance evaluation tests. The implementation outperforms existing competitors, e.g. Supremica [1], CTCT [16].

There are several directions to improve the presented results. We admit that BDDs allow one to solve the problems of large size. However, even their computation power is not enough for addressing real-world (industrial) problems, which are usually huge in terms of the number of plant's states and have hundreds of requirements of different levels. Thus, we believe that a major evolution of this work is to leverage it with abstraction (hierarchical) techniques and/or modular synthesis procedures.

The other direction is to extend (enrich) the specification language, for example, with preferences, recovery specifications, etc. (see EaGLe [8]). Another direction is to face the partial observability challenge. Finally, there is space for improving the implementation, for example, investigate the use of IDDs in place of BDDs.

## References


[1] K. Åkesson, M. Fabian, H. Flordal, and A. Vahidi, "Supremica - A Tool for Verification and Synthesis of Discrete Event Systems," in *Proc. of the* $11^{th}$ *Mediterranean Conference on Control and Automation*, June 2003.

[2] M. Barbeau, F. Kabanza, and R. St-Denis, "A method for the synthesis of controllers to handle safety, liveness, and real-time constraints," *IEEE Transactions on Automatic Control*, vol. 43, no. 11, pp. 1543–1559, November 1998.

[3] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, "MBP: a Model Based Planner," in *Proc. of IJCAI'01 workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic Model Checking: $10^{20}$ States and Beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, June 1992.

[5] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Kluwer Academic Publ., 1999.

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," 2002. [Online]. Available: citeseer.nj.nec.com/508675.html

[7] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[8] U. Dal Lago, M. Pistore, and P. Traverso, "Planning with a Language for Extended Goals," in *Proc. of the* $18^{th}$ *National Conference on Artificial Intelligence and* $14^{th}$ *Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, Aug. 2002, pp. 447–454.

[9] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier, 1990.

[10] J. Huang and R. Kumar, "Nonblocking Directed Control of Discrete Event Systems," in *Proc. of the* $44^{th}$ *IEEE Conference on Decision and Control and the European Control Conference*, Dec. 2005, pp. 7627–7632.

[11] S. Jiang and R. Kumar, "Supervisory Control of Discrete Event Systems with CTL* Temporal Logic Specications," in *Proc. of the* $40^{th}$ *IEEE Conference on Decision and Control*, December 2001. [Online]. Available: citeseer.nj.nec.com/447912.html

[12] M. Maidl, "The common fragment of CTL and LTL," in *IEEE Symposium on Foundations of Computer Science*, 2000, pp. 643–652. [Online]. Available: citeseer.ist.psu.edu/maidl00common.html

[13] P. Ramadge and W. Wonham, "The control of discrete event systems," in *Proc. of IEEE*, vol. 77 (1), 1989, pp. 81–98.

[14] K. T. Seow, M. Gai, and T. L. Lim, "A temporal logic specification interface for automata-theoretic finitary control synthesis," in *Proc. of the IEEE International Conference on Robotics and Automation*, Apr. 2005.

[15] A. Vahidi, B. Lennartson, and M. Fabian, "Efficient analysis of large discrete-event systems with binary decision diagrams," in *Proc. of the* $44^{th}$ *IEEE Conference on Decision and Control and the European Control Conference*, Dec. 2005, pp. 2751–2756.

[16] W.M.Wonham, "Notes on Control of Discrete-Event Systems." [Online]. Available: http://www.control.utoronto.ca/people/profs/wonham/wonham.html

[17] Z. Zhang and W. Wonham, "STCT: An Efficient Algorithm for Supervisory Control Design," in *Proc. of the Symposium on Supervisory Control of Discrete Event Systems (SCODES2001)*, June 2001. [Online]. Available: citeseer.nj.nec.com/zhang01stct.html