Discrete Event Dyn Syst (2017) 27:209–260
DOI 10.1007/s10626-015-0223-0

Supervisory control and reactive synthesis: a comparative introduction

Rüdiger Ehlers[1,2] · **Stéphane Lafortune**[3] · **Stavros Tripakis**[4,5] · **Moshe Y. Vardi**[6]

Received: 3 July 2014 / Accepted: 8 December 2015 / Published online: 15 March 2016

Abstract This paper presents an introduction to and a formal connection between synthesis problems for discrete event systems that have been considered, largely separately, in the two research communities of *supervisory control* in control engineering and *reactive synthesis* in computer science. By making this connection mathematically precise in a paper that attempts to be as self-contained as possible, we wish to introduce these two research areas to non-expert readers and at the same time to highlight how they can be bridged in the context of classical synthesis problems. After presenting general introductions to supervisory control theory and reactive synthesis, we provide a novel reduction of the basic supervisory control problem, non-blocking case, to a problem of reactive synthesis with plants and with a maximal permissiveness requirement. The reduction is for fully-observed systems that are controlled by a single supervisor/controller. It complements prior work that has explored

This work was primarily supported by the US National Science Foundation (NSF) Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering (grant CCF-1138860). Additional support from the following sources is also acknowledged: the University of California at Berkeley, the University of Michigan, Rice University, Aalto University, the iCyPhy Research Center (supported by IBM and United Technologies), the Academy of Finland, NSF award CNS-1329759, and the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative.

✉ Stéphane Lafortune
stephane@umich.edu

[1] University of Bremen, Bremen, Germany

[2] DFKI GmbH, Bremen, Germany

[3] University of Michigan, Ann Arbor, MI, USA

[4] University of California at Berkeley, Berkeley, CA, USA

[5] Aalto University, Helsinki, Finland

[6] Rice University, Houston, TX, USA

problems at the interface of supervisory control and reactive synthesis. The formal bridge constructed in this paper should be a source of inspiration for new lines of investigation that will leverage the power of the synthesis techniques that have been developed in these two areas.

Keywords Supervisory control · Reactive synthesis · Non-blockingness · Maximal permissiveness

1 Introduction

The goal of this paper is to introduce and present a formal connection between synthesis problems that have been considered, largely separately, in the two research communities of control engineering and formal methods. By making this connection mathematically precise in a paper that attempts to be as self-contained as possible, we hope to introduce these two research areas to non-expert readers and at the same time inspire future research that will further bridge the gap between them and leverage their complementarity. Given the pedagogical nature of this paper, we focus on connecting the most basic synthesis problem in supervisory control, for safety and non-blockingness in the case of full observation, with a suitably-defined analogous problem in the field of reactive synthesis. We start our discussion with a general introduction to the two research areas under consideration.

1.1 Supervisory control of discrete event systems

Feedback control of dynamic systems is an essential element of our technological society. Control theory was originally developed for systems with continuous variables that evolve in time according to dynamics described by differential or difference equations. Since the 1980s, the field of Discrete Event Systems (DES) in control engineering has been concerned with the application of the feedback paradigm of control theory to the class of dynamical systems with discrete state space and event-driven dynamics.

The DES community has been investigating feedback control of DES using models from computer science, such as automata and Petri nets. The body of control theory developed in DES has been for specifications that are expressible as regular languages, in the case of DES modeled by automata, or in terms of constraints on the state (marking vector), in the case of DES modeled by Petri nets. Control-theoretic frameworks have been developed for both of these modeling formalisms. In this paper, we focus on the supervisory control theory for systems modeled by finite-state automata and subject to regular language specifications. Both the plant and the specification are represented as finite-state automata over a common event set. The foundations for this framework were developed in the seminal work of Ramadge and Wonham (1987), Wonham and Ramadge (1987). Since then, a whole body of theory has been developed that covers a wide variety of control architectures and information structures, with vertical and horizontal modularity. The reader is referred to Cassandras and Lafortune (2008), Wonham (2015) for textbook expositions of this theory; other relevant references are the monograph (Kumar and Garg 1995), the survey papers (Ramadge and Wonham 1989; Thistle 1996), and the recent edited book (Seatzu et al. 2013). The focus of this theory is on the synthesis of provably safe and non-blocking controllers for a given uncontrolled system, or *plant* in control engineering terminology, despite limited actuation and limited sensing capabilities. In automated manufacturing applications for instance, the plant would be the joint operation of a set of robots, conveyors, Automated Guided Vehicles (AGVs), and numerically controlled machines, and the controller would be implemented using one or more Programmable Logic Controllers (PLCs). Safety properties are expressed in terms of bad states where robots and/or AGVs collide for instance, or bad sequences of events that correspond to incorrect assembly for instance. Non-blockingness captures the fact that product assembly should be completed in its entirety, followed by a return of all components to their initial states.

1.2 Reactive synthesis

It is widely acknowledged that many design defects originate in the failure of the implementation to accurately capture the designer's intent. Underlying the reactive synthesis approach is the realization that many requirements can be expressed as formal *temporal assertions*, capturing intended system functionality in a declarative fashion. Assertions can express both *safety* properties, such as "a Grant is always followed by Busy", and *liveness* properties, such as "a Request is eventually followed by a Grant". Thus, the *functional specification* of a system can be expressed as a set of temporal assertions.

The *assertion-based approach to system specification* underlays early work on program verification (Francez 1992), whose focus was on input/output properties. This was later extended to temporal properties of ongoing computations (Pnueli 1977), which enabled the application of formal verification techniques to reactive systems–systems that have ongoing interactions with their environments (Harel and Pnueli 1985). One of the most successful applications of the assertion-based approach has been *model checking*, an algorithmic formal-verification technique (Clarke and Emerson 1981; Clarke et al. 1986; Lichtenstein and Pnueli 1985; Queille and Sifakis 1982; Vardi and Wolper 1986); see (Clarke et al. 2000) for an in depth coverage.

The design of reactive systems, systems that engage in an ongoing interaction with their environment, is one of the most challenging problems in computer science (Harel and Marelly 2003; Harel and Pnueli 1985). The assertion-based approach constitutes a significant progress towards addressing this challenge. While there has been impressive recent progress on applying formal methods in verification (Jackson 2009), in current design methodology, design and verification are distinct phases, typically executed by separate teams. Substantial resources are spent on verifying that an implementation conforms to its specifications, and on integrating different components of the system. Not only do errors discovered during this phase trigger a costly reiteration of design and programming, but more importantly, verification offers only quality control, not quality improvement, and hence, current design methodology does not produce systems that are safe, secure, and reliable.

Currently, when formal assertions are being used, it is in testing and verification, *after* a significant effort has already gone into the development effort. When errors are found, significant effort has to be expended on design change. An old dream in computer science is that of *design automation*, in which the process of converting formal specification to implementation is, to a major extent, automated. The implication is that a major portion of the manual design effort should go into the development of high-level specifications, since much of the implementation effort can then be automated. The technique of compiling high-level formal requirements to low-level system code is referred to as *synthesis*, and was proposed already in Church (1957), Green (1969). Follow up work in Büchi and Landweber (1969), Rabin (1972) addressed the problem mathematically, but it seemed quite far from being applicable to real-life problems.

In the late 1980s, several researchers realized that the classical approach to system synthesis (Green 1969), where a system is extracted from a proof that the specification is satisfiable, is well suited to *closed* systems (systems with no inputs), but not to *reactive* systems. In reactive systems, the system interacts with the environment, and a correct system should then satisfy the specification no matter how the environment behaves, i.e., no matter which inputs the environment provides to the system. If one applies the techniques of Emerson and Clarke (1982), Manna and Wolper (1984) to reactive systems, one obtains systems that are correct only with respect to *some* environment behaviors. Pnueli and Rosner (1989a), Abadi et al. (1989), and Dill (1989) argued that the right way to approach synthesis of reactive systems is to use the model of a, possibly infinite, game between the environment and the system. A correct system can be then viewed as a winning strategy in this game. It turns out that satisfiability of the specification is not sufficient to guarantee the existence of such a strategy. Abadi et al. called specifications for which a winning strategy exists *realizable*. Since then, the subject of *reactive synthesis* has been an active area of research, attracting a considerable attention, for example (Kupferman and Vardi 2000; Pnueli and Rosner 1989b; Vardi 1995; Wong-Toi and Dill 1991).

1.3 Contributions and organization of this paper

We start in Section 2 by presenting technical introductions to supervisory control (Section 2.1) and reactive synthesis (Section 2.2). We have tried to make Section 2 as self-contained as possible, so that readers who are not experts in either domain could follow the ensuing results connecting these two areas. However, these introductions are not meant to be thorough and readers interested in learning these areas in more depth should consult the references mentioned above.

The main results connecting supervisory control and reactive synthesis are contained in Section 3. That section is organized into several parts. First, we present in Section 3.1 a simplification of the basic supervisory control problem, non-blocking version, to one where the safety specification has been absorbed into the plant model. We then show that the resulting Simple Supervisory Control Problem (SSCP) has a state-based solution. The results on SSCP will facilitate bridging the gap in the remainder of Section 3. Second, for bridging reactive synthesis with supervisory control, we need two technical steps: the first step is to consider reactive synthesis with plants; the second step is to bring in the issue of maximal permissiveness into this reactive synthesis setting. These two steps are covered in Section 3.2. With the above technical results established, we establish the formal reduction from SSCP to a reactive synthesis problem with plants and maximal permissiveness in Section 3.3. This formal reduction is the main technical contribution of this paper. Section 3.4 discusses algorithmic issues and Section 3.5 discusses links between the reactive synthesis problem with plants and the more standard reactive synthesis problem without plants.

Finally, some concluding comments and directions for future work are given in Section 4. A preliminary and partial version of this paper, without proofs, appears in Ehlers et al. (2014).

1.4 Related works

This paper is not the first to explore connections between supervisory control and reactive synthesis. On the supervisory control side, several authors have considered control of discrete event systems subject to temporal logic specifications; see, e.g.,Thistle and Wonham 1986, Lin (1993), Jiang and Kumar (2006). Supervisory control of discrete event systems with infinite behavior, i.e., modeled by languages over E^{ω} instead of E^* for a given event set E, has also been considered by many researchers; see, e.g., Ramadge (1989), Kumar et al. (1992), Thistle and Wonham (1994a), Thistle and Wonham (1994b), Thistle (1995), Thistle and Malhamé (1998), Lamouchi and Thistle (2000). On the other hand, several researchers in the formal methods community have investigated supervisory control of fully- and partially-observed discrete event systems, in untimed, timed, and hybrid system settings; see, e.g., Hoffmann and Wong Toi (1992), Maler et al. (1995), Asarin et al. (1995), Henzinger and Kopke (1997), Kupferman et al. (2000), Madhusudan (2001), Cassez et al. (2002), Arnold et al. (2003), Riedweg and Pinchinat (2003). Researchers from both the supervisory control and formal methods communities have also studied problems of distributed or decentralized controller synthesis, where more than one controllers are (simultaneously) synthesized, e.g., see Pnueli and Rosner (1990), Rudie and Wonham (1992), Barrett and Lafortune (1998), Lamouchi and Thistle (2000), Overkamp and van Schuppen (2000), Ricker and Rudie (2000), Tripakis (2004), Lustig and Vardi (2009), Komenda et al. (2012), Seatzu et al. (2013).

In the present paper, we restrict attention to the basic problem of centralized supervisory control for fully-observed systems modeled by languages of finite strings. Our goal is to establish a precise connection of this classical work with problems of reactive synthesis, by showing how specific problem instances reduce to each other. While more advanced connections have been discussed in the above-cited references, our goal is pedagogical in nature and aims at establishing a bridge between these two areas at the most elementary level. Still, to the best of our knowledge, the formal reduction presented in Section 3 has not been published elsewhere. Our results therefore complement existing works from a technical standpoint.

2 Classical frameworks

In this section, we present technical introductions to the fields of *supervisory control* and *reactive synthesis*. These introductions are not meant to be comprehensive, but rather their goal is to allow readers who are not experts in either or both of these fields to follow the later results in Section 3 on establishing a precise connection between basic synthesis problems in these two fields.

2.1 Supervisory control

In supervisory control of Discrete Event Systems (DES), the system to be controlled, i.e., the plant, is typically modeled as a set of interacting finite-state automata coupled by common events or as a Petri net. In order to obtain a monolithic model that will be used for analysis and control synthesis purposes, the parallel composition of the set of interacting automata is performed or the reachability graph of the Petri net is constructed. We restrict attention to plants with finite state spaces. Also, we assume full event observability.

Let the plant be denoted by G. (Formal definitions will follow.) G captures the entire set of possible behaviors of the plant and it is called the "uncontrolled system." In general, some of this behavior is not acceptable in the sense that it is not *safe* with respect to a given *specification* or that it results in deadlock or livelock. Consequently, we wish to restrict the behavior of G by means of a *feedback controller*, or *supervisor* in DES terminology. The standard feedback loop of supervisory control theory is shown in Fig. 1. The "input" to the supervisor S is the string of events generated so far by G. All the events of G are observed

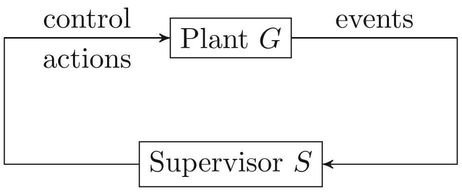


Fig. 1 Closed-loop system S/G, where S issues control actions in response to the events generated by G

by S. The "output" of S is a control action that tells G which event(s) it is allowed to do at that moment. The supervisor may allow more than one event, in which case the system will decide which allowed event to execute next. The mechanism by which G chooses which allowed event to execute next is not modeled. One may think of G as a semi-autonomous system for instance. In general, the supervisor may not have full actuation capabilities, i.e., there may be events of G that it cannot disable. These events are called *uncontrollable*. How to deal with uncontrollable events is one of the contributions of supervisory control theory.

Example 1 (Coffee Machine) For the sake of illustration, assume that our plant G is a coffee machine that can grind coffee beans, brew coffee, and deliver a cup of coffee. Its interface with the user is a "coffee button" that generates an event, denoted by c, when it is pressed by the user. The automaton representation of that machine is shown in Fig. 2. We give the formal definition of an automaton below; for now, we explain the transition structure. The initial state is 1 (this is denoted by the fact that state 1 has an incoming arrow). State 1 is also an accepting or *marked* state (denoted by the fact that the state is drawn as a double circle). Upon occurrence of event c, the coffee machine moves to a new state in which it can execute an arbitrary number of "grind" events, which are denoted by g, as well as an arbitrary number of "brew" events, which are denoted by b; these events self-loop at state 2. Finally, when grinding and brewing are completed, the coffee is delivered (poured in cup) and the machine returns to its initial state; this is represented by event r. For simplicity, we assume that the machine ignores further pressing of the coffee button while it is grinding and brewing, i.e., until event r occurs. This is modeled by the self-loop for event c at state 2.

G represents the physical capabilities of the machine, without an appropriate control protocol, i.e., without a *specification*. As given, this behavior is *unsafe*: first, brewing should always be preceded by grinding; second, no grinding should occur after brewing has started.

Moreover, one may wish to follow special coffee recipes, parameterized by the number of g events (more events means finer ground coffee) and the number of b events (more events for stronger coffee). One such recipe could be that coffee is prepared by one grinding step followed by two brewing steps. Another recipe for stronger coffee could call for two grinding steps followed by three brewing steps.

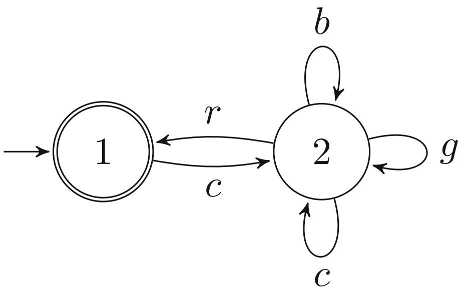


Fig. 2 Automaton G: uncontrolled coffee machine

We wish to synthesize a supervisor S that will restrict the behavior of G in order to satisfy the above safety constraints and allow the two possible recipes. In this example, event c is uncontrollable, as the supervisor cannot tell the plant to ignore a request for coffee when it is in state 1. (Note that such requests are ignored while the plant is in state 2, as captured in the model in Fig. 2.) The other events are assumed to be controllable, as the coffee machine has actuators for grinding, brewing, and delivering a cup of coffee; hence, the control protocol can decide when to activate or de-active these actuators. For instance, after the occurrence of event c, S needs to tell G that it should execute a g event, not a b event, as grinding must precede brewing. Hence, immediately after event c occurs, S should *disable* event b. Similarly, after the first b event, S should disable event g until the next cup is prepared. As for event r, it should remain disabled until a recipe is completed. Here, in the spirit of supervisory control, we assume that the supervisor will not force one of the allowed recipes. Namely, after one occurrence of g, S can enable both g and b and let the plant “randomly” choose to do either another g event or to start brewing, i.e., randomly choose which recipe to implement (perhaps on the basis of some other features not included in this simple model). If the plant chooses to execute b, then further occurrences of g will be disabled by S. On the other hand, after two consecutive occurrences of event g, S needs to tell G that it should not execute any more g events for the current cup under preparation, as no recipe calls for three grinding steps; i.e., S must disable the third consecutive occurrence of event g. When the plant has completed either recipe, S disables both g and b and enables r in order to allow the plant to deliver the cup of coffee.

In supervisory control, the objective is to automatically synthesize a supervisor S that provably satisfies all given specifications. The inputs to the synthesis process are: (i) the model G; (ii) the sets of controllable and uncontrollable events; and (iii) an automaton model of the specifications imposed on G. In the rest of this section, we present the main concepts needed before we can proceed to bridging the gap with reactive synthesis.

2.1.1 Plant model

In supervisory control theory, plants are typically modeled as deterministic finite-state automata. A deterministic finite-state automaton (DFA) is a 5-tuple

$$G = (X, x_0, X_m, E, \delta)$$

where

- X is a finite set of states, $x_0 \in X$ is the initial state, and $X_m \subseteq X$ is the set of *marked* (i.e., *accepting*) states;
- E is a finite set of events. E is (implicitly) partitioned into two disjoint subsets:

$$E = E_c \cup E_{uc}$$

  where E_c models the set of *controllable* events and E_{uc} the set of *uncontrollable* events.
- $\delta : X \times E \to X$ is the transition function, which in general will be partial.

The reason for the transition function to be partial is the fact that G models the physically possible behavior of a DES, as a generator of events. Selection of the states to “mark,” i.e., to be included in X_m, is a modeling consideration to capture strings that represent that the system has completed some task. In many systems with cyclic behavior, such as our coffee machine, the initial state is marked, as returning to it means that a task has been completed and a new one can start.

A state $x \in X$ is called a *deadlock state* if for all $e \in E$, $\delta(x, e)$ is undefined.

It is useful to "lift" the transition function δ to a function

$$\delta^* : X \times E^* \to X$$

defined as follows:

$$\begin{aligned} \delta^*(x, \varepsilon) &= x \\ \delta^*(x, \sigma \cdot e) &= \delta(\delta^*(x, \sigma), e) \end{aligned}$$

where ε denotes the empty string and $\cdot$ denotes string concatenation. (We usually omit writing $\cdot$ unless needed for clarity of notation.) Note that since δ is partial, δ^* is also partial. Since we always extend δ to δ^*, we shall drop the "*" superscript hereafter and simply refer to the extended function as δ.

The DES G defines the following languages:

$$\mathcal{L}(G) = \{\sigma \in E^* \mid \delta(x_0, \sigma) \text{ is defined}\} \qquad \text{and} \qquad \mathcal{L}_m(G) = \{\sigma \in E^* \mid \delta(x_0, \sigma) \in X_m\}.$$

Given $K \subseteq E^*$, let $\overline{K}$ denote the *prefix-closure* of K:

$$\overline{K} = \{\sigma \mid \exists \sigma' \in E^* : \sigma\sigma' \in K\}$$

In the definition above σ' can be the empty string, so $K \subseteq \overline{K}$ for all K.

Moreover, for any DES G:

$$\mathcal{L}_m(G) \subseteq \overline{\mathcal{L}_m(G)} \subseteq \mathcal{L}(G) = \overline{\mathcal{L}(G)}$$

Note that $\overline{\mathcal{L}_m(G)} \neq \mathcal{L}(G)$ in general. This is because $\mathcal{L}(G)$ may contain strings that cannot be extended to yield strings in $\mathcal{L}_m(G)$. In other words, G may contain reachable states that cannot reach any marked state.

2.1.2 Supervisors

A *supervisor* for G is a function $S : E^* \to 2^E$. It reads a string σ representing the history of what has happened so far and returns the set of controllable events that are allowed to occur. To ensure that S never disables an uncontrollable event, we require that $E_{uc} \subseteq S(\sigma)$ for all $\sigma \in E^*$ (alternatively, we could also define S to be a function $S : E^* \to 2^{E_c}$). Note that $S(\sigma) \cap E_c$ may be empty.

Sometimes S is required to satisfy the following property:

$$\forall \sigma, c \in E_c : c \in S(\sigma) \implies \delta(x_0, \sigma c) \text{ is defined} \tag{1}$$

which states that S allows a controllable event e only if e is feasible in G. This is not an essential requirement on S: we can simply ignore controllable events that S allows but are not feasible in G. The same comment applies to enabled but infeasible uncontrollable events.

When S is a supervisor specifically designed for G, the history σ can only be generated by G, therefore S can also be defined as a function

$$S : \mathcal{L}(G) \to 2^E$$

However, in general, it is more convenient to define S to be a function over E^*, since this allows us to use the same supervisor for different plants, as long as E_c and E_{uc} remain the same.

Remark 1 (Full observability) In the current framework, a plant is *fully observable* by a supervisor, for two reasons. First, the supervisor, defined as a function with domain E^*, observes the entire sequence of events generated by the plant (in partial observability frameworks, only a subset of events are observed). Second, the plant itself is a deterministic automaton. Therefore, given the sequence of observed events, the supervisor can uniquely determine the current state of the plant.

2.1.3 Closed-loop system

Given a plant $G = (X, x_0, X_m, E, \delta)$ and a supervisor $S : E^* \to 2^E$ for G, the *closed-loop system* S/G, according to the feedback loop in Fig. 1, is a DES that is formally defined as follows:

$$S/G = (X', x'_0, X'_m, E, \delta')$$

where

- $X' = X \times \mathcal{L}(G)$
- $x'_0 = (x_0, \varepsilon)$
- $X'_m = X_m \times \mathcal{L}(G)$
- $\delta'((x, \sigma), e) = \begin{cases} (\delta(x, e), \sigma e) & \text{if } \delta(x, e) \text{ is defined and } e \in S(\sigma) \\ \text{undefined} & \text{otherwise.} \end{cases}$

A state in S/G is a pair (x, σ) where $x \in X$ is a state of the plant G and $\sigma \in \mathcal{L}(G)$ is the history observed so far. Thus, S/G is an infinite-state automaton, except for the special case that G is loop-free. This need not worry us for now. At this point, we are mainly interested in defining the synthesis problem, and not the algorithm to solve it. The initial state of S/G is (x_0, ε), since x_0 is the initial state of G and the history is initially empty. X'_m is defined as $X_m \times \mathcal{L}(G)$, meaning that a behavior of the closed-loop system is marked iff it is marked by G. In other words, we only consider supervisors that do not affect the marking of states in the plant. The definition of the transition function δ' is explained as follows. Given current state (x, σ) and event e:

- when $\delta(x, e)$ is undefined (i.e., the plant does not have a transition from x for e), then $\delta'((x, \sigma), e)$ is also undefined
- otherwise, assuming $\delta(x, e) = x'$,
  - if e is uncontrollable, i.e., $e \in E_{uc}$, then e is allowed by the supervisor and the next state is $(x', \sigma e)$, i.e., the plant moves to x' and the supervisor observes e,
  - if e is controllable and allowed by the supervisor S, i.e., $e \in E_c \cap S(\sigma)$, then the next state is again $(x', \sigma e)$,
  - otherwise (i.e., if $e \in E_c \setminus S(\sigma)$, meaning that e is controllable but not allowed by S), $\delta'((x, \sigma), e)$ is undefined. This is the only case when an event e which is allowed in G is forbidden in the closed-loop system.

Note that when $\delta(x, e)$ is defined, $\sigma \in \mathcal{L}(G)$ implies $\sigma e \in \mathcal{L}(G)$. This ensures that if (x, σ) is a valid state of S/G, i.e., $(x, \sigma) \in X'$, then $\delta'((x, \sigma), e)$ is also a valid state of S/G, so that the state-space of S/G is well defined.

S/G is an automaton (albeit an infinite-state one), therefore, languages $\mathcal{L}(S/G)$ and $\mathcal{L}_m(S/G)$ are defined as stated above. Note that, by definition, S/G is a restriction of G, therefore,

$$\mathcal{L}(S/G) \subseteq \mathcal{L}(G) \quad \text{and} \quad \mathcal{L}_m(S/G) \subseteq \mathcal{L}_m(G)$$

Moreover, it is easy to verify that

$$\mathcal{L}_m(S/G) = \mathcal{L}(S/G) \cap \mathcal{L}_m(G)$$

since a marking in S/G is completely determined by a marking in G. When S is applied to G as described above, the definition of $S(\sigma)$ for $\sigma \in E^* \setminus \mathcal{L}(S/G)$ is irrelevant, since the controlled behavior will never exceed $\mathcal{L}(S/G)$.

As an example, consider the plant G_1 shown in Fig. 3. Let $E_c = \{c_1, c_2\}$ and $E_{uc} = \{u\}$ (we generally use the convention that events $c, c', c_1, c_2, \ldots$ are controllable, while events $u, u', u_1, u_2, \ldots$ are uncontrollable). Consider two supervisors S_1 and S_2 for G_1, defined as follows:

$$S_1(\sigma) = \{c_1, u\} \text{ for all } \sigma \quad S_2(\sigma) = \begin{cases} \{c_1, u\} & \text{if } \sigma = \varepsilon \\ \{c_2, u\} & \text{otherwise} \end{cases}$$

The closed-loop systems S_1/G_1 and S_2/G_1 are shown in Fig. 3. For simplicity, states in the closed-loop systems are labeled as in the original plant G_1, instead of being labeled as pairs (x_0, ε), (x_1, c_1), (x_3, u), and so on.

Remark 2 (Supervisors vs. controllers) Supervisory control theory typically uses the term “supervisor” instead of “controller”. The term “supervisor” is well-chosen because in this framework supervisors are like “parents”: they can disable options, but they cannot “make things happen”. For instance, a supervisor cannot force the plant to take a certain transition, even when this transition is controllable. The supervisor can only allow a controllable transition. If this is the only outgoing transition from the current state, then presumably this will happen (although the state may be marked, with the interpretation that the plant “stops” there). But if there are multiple (controllable or uncontrollable) transitions from that state, the plant could choose any of them, without the supervisor having any control over this choice.

2.1.4 A trivial synthesis problem

A supervisor is needed because without it the plant may generate illegal behaviors. The supervisor aims at restricting the plant’s behaviors, so that they are all contained in a set of “good”, or “legal” behaviors.

One way to formalize this idea is to assume that we are given a language of “good” behaviors, L_{am}, called the *admissible marked language*. Then we could define a synthesis problem where we ask for a supervisor S (if it exists) such that $\mathcal{L}_m(S/G) \subseteq L_{am}$. This, however, is not a very interesting problem, for a number of reasons.

First, in terms of synthesis, the problem is trivial. Indeed, instead of *searching* for an arbitrary supervisor S, it suffices to simply *check* whether the *most-restrictive* (or *least-permissive*) supervisor works. The most-restrictive supervisor S_{mr} is the supervisor that

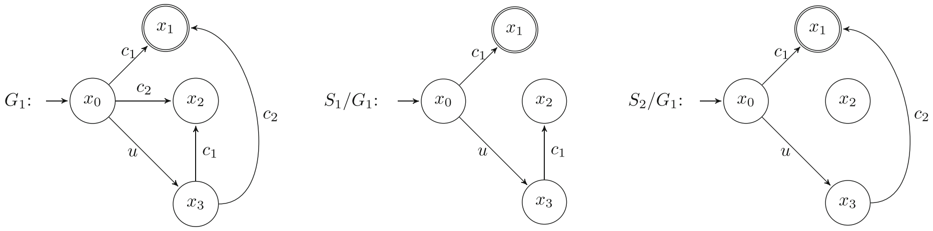


Fig. 3 Plant G_1 and two closed-loop systems

disables everything that it can disable, i.e., $S_{mr}(\sigma) = E_{uc}$, for any σ. If S_{mr} satisfies $\mathcal{L}_m(S_{mr}/G) \subseteq L_{am}$, then we have found a solution to the above synthesis problem. Otherwise, it is easy to see that no solution exists. Indeed, any other supervisor S is bound to allow more behaviors than S_{mr}, that is, $\mathcal{L}_m(S_{mr}/G) \subseteq \mathcal{L}_m(S/G)$. Therefore, if $\mathcal{L}_m(S_{mr}/G)$ is not a subset of L_{am}, neither can $\mathcal{L}_m(S/G)$ be.

The fact that this problem is trivial (indeed, it is not really a synthesis problem, but a verification problem) should not necessarily deter us. On the contrary, the easier the problem, the better. However, the second and most important reason why the above problem is not the right one is the fact that the most-restrictive supervisor is rarely what we want. Indeed, the most-restrictive supervisor may be far too restrictive. It may, for example, introduce deadlocks. For the plant G_1 in Fig. 3, S_{mr} will result in a new deadlock at state 3 of G_1, since the only events out of that state are controllable events that are disabled by S_{mr}. We therefore need a richer way to specify desirable supervisors. Toward this goal, we introduce next the notion of non-blockingness.

2.1.5 Non-blockingness

Let G be a plant and S a supervisor for G. S is said to be *non-blocking for G* iff

$$\overline{\mathcal{L}_m(S/G)} = \mathcal{L}(S/G).$$

Note that, as mentioned above, $\overline{\mathcal{L}_m(S/G)} \subseteq \mathcal{L}(S/G)$ always holds. Therefore, non-blockingness is equivalent to $\mathcal{L}(S/G) \subseteq \overline{\mathcal{L}_m(S/G)}$. Non-blockingness says that the closed-loop system should not contain behaviors that cannot be extended to marked behaviors. More precisely, there should be no deadlock states that are not marked, and there should be no absorbing strongly connected components that do not contain a marked state; the latter situation corresponds to a *livelock.*

As an example, consider again plant G_1 and supervisors S_1, S_2 of Fig. 3. It can be seen that S_1 is blocking since ($uc_1 \in \mathcal{L}(S_1/G)$ but $\overline{\mathcal{L}_m(S_1/G)} = \{\varepsilon, c_1\}$); on the other hand, S_2 is non-blocking.

The following is a useful characterization of non-blockingness. Its proof is straightforward from the definition of non-blockingness.

Lemma 1 *S is a non-blocking supervisor for G iff from every reachable state of S/G there is a path to a marked state of S/G.*

2.1.6 Safety properties and admissible marked languages

Before we can give a formal statement of the basic supervisory control problem defined below (in Section 2.1.8), we need to formalize the notions of admissible (marked) language and of maximal permissiveness. This is done in this and the next subsections.

An admissible marked language in supervisory control, denoted L_{am}, captures the set of "legal", or "good", behaviors. Typically, the designer first comes up with a *prefix-closed* regular language L_a (i.e., a regular language such that $\overline{L_a} = L_a$). L_a captures the set of *safe* behaviors. For *safety* properties, such sets are prefix-closed, since for every unsafe behavior σ, every extension $\sigma \cdot \sigma'$ of σ is also unsafe. Conversely, if σ is safe, every prefix of σ is also safe. Once L_a is defined, the designer typically sets L_{am} to be the intersection $L_{am} := L_a \cap \mathcal{L}_m(G)$, so that L_{am} captures the set of all safe behaviors that can be generated and are marked by the plant. L_{am} has two useful properties:

1. $L_{am} \subseteq \mathcal{L}_m(G)$ (by definition, since $L_{am} = L_a \cap \mathcal{L}_m(G)$).
2. L_{am} is "$\mathcal{L}_m(G)$-closed". Given languages K and L with $K \subseteq L \subseteq E^*$, we say that K *is L-closed* iff

$$K = \overline{K} \cap L.$$

   Notice that since $K \subseteq L$ and $K \subseteq \overline{K}$, $K \subseteq \overline{K} \cap L$ always holds. Therefore requiring L-closure is requiring that $\overline{K} \cap L \subseteq K$. Then, it is easy to see that if $L_{am} = L_a \cap \mathcal{L}_m(G)$ then $\overline{L_{am}} \cap \mathcal{L}_m(G) \subseteq L_{am}$.

In the sequel we will assume that the given admissible marked language L_{am} satisfies conditions 1 and 2 above, i.e., $L_{am} \subseteq \mathcal{L}_m(G)$ and L_{am} is $\mathcal{L}_m(G)$-closed. Moreover, we will assume that L_{am} is regular and we will be using a finite-state automaton to represent it.

Given DES G and $L_{am} \subseteq \mathcal{L}_m(G)$, a supervisor S is said to be *safe for G with respect to* L_{am} if $\mathcal{L}_m(S/G) \subseteq L_{am}$, i.e., the only marked strings allowed under control are safe marked strings. We often drop the reference to L_{am} and simply say that a supervisor is "safe" if it is clear from the context which safety specification is being referred to.

As an example, consider plant G_1 of Fig. 3. Let $L_2 = \{uc_2\}$, which can be interpreted as capturing the safety property "c_1 should never occur". (As explained above, L_2 is obtained by taking the intersection of $\mathcal{L}_m(G_1)$ with the prefix-closed language containing all strings not including c_1, i.e., ε, u, c_2, uu, c_2c_2, and so on.) L_2 is $\mathcal{L}_m(G_1)$-closed. L_2 is also a strict subset of $\mathcal{L}_m(G_1)$. Therefore, L_2 is a valid admissible marked language. Consider the supervisor S_3 defined as follows:

$$S_3(\sigma) = \begin{cases} \{u\} & \text{if } \sigma = \varepsilon \\ \{c_2, u\} & \text{otherwise} \end{cases}$$

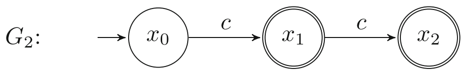
It can be seen that S_3 is non-blocking for G_1 and satisfies $\mathcal{L}_m(S_3/G_1) \subseteq L_2$; therefore, S_3 is safe w.r.t. L_2.

Remark 3 $L_{am} \subseteq \mathcal{L}_m(G)$ is not a restrictive assumption, even for arbitrary L_{am}, i.e., not necessarily satisfying $L_{am} = L_a \cap \mathcal{L}_m(G)$. If $L_{am} \not\subseteq \mathcal{L}_m(G)$, we can set $L'_{am} := L_{am} \cap \mathcal{L}_m(G)$ (thus achieving the condition $L'_{am} \subseteq \mathcal{L}_m(G)$) and ask for a supervisor such that $\mathcal{L}_m(S/G) \subseteq L'_{am}$. Any such supervisor S also satisfies $\mathcal{L}_m(S/G) \subseteq L_{am}$, since $L'_{am} \subseteq L_{am}$. Conversely, any supervisor which satisfies $\mathcal{L}_m(S/G) \subseteq L_{am}$ also satisfies $\mathcal{L}_m(S/G) \subseteq L'_{am}$, since $\mathcal{L}_m(S/G) \subseteq \mathcal{L}_m(G)$. Therefore, asking for a supervisor such that $\mathcal{L}_m(S/G) \subseteq L_{am}$ is equivalent to asking for a supervisor such that $\mathcal{L}_m(S/G) \subseteq L'_{am}$. Thus, we can assume $L_{am} \subseteq \mathcal{L}_m(G)$ without loss of generality.

On the other hand, an arbitrary L_{am} is not necessarily $\mathcal{L}_m(G)$-closed, as the following example illustrates.

Consider the plant G_2 shown in Fig. 4, where $\mathcal{L}_m(G_2) = \{c, cc\}$. Let $L_1 = \{cc\}$. L_1 is not $\mathcal{L}_m(G_2)$-closed. Indeed, $c \in \overline{L_1} \cap \mathcal{L}_m(G_2)$ but $c \notin L_1$. It is easy to find a blocking supervisor that ensures $\mathcal{L}_m(S/G_2) \subseteq L_1$. In fact, the most-restrictive supervisor S_{mr} achieves $\mathcal{L}_m(S_{mr}/G_2) = \emptyset \subseteq L_1$. This supervisor is blocking, because ε is in $\mathcal{L}(S_{mr}/G_2)$ but not in $\overline{\mathcal{L}_m(S_{mr}/G_2)} = \emptyset$. A non-blocking supervisor S that ensures $\mathcal{L}_m(S/G_2) \subseteq L_1$

Fig. 4 Plant G_2



does not exist. Indeed, S can only disable transitions, and it cannot disable both transitions of G_2 because this would be blocking. Thus, at least one of c or cc must be in $\mathcal{L}_m(S/G_2)$. But $c \in \mathcal{L}_m(S/G_2)$ is not allowed, because this would imply $\mathcal{L}_m(S/G_2) \not\subseteq L_1$. Therefore it must be that $\mathcal{L}_m(S/G_2) = \{cc\}$. But it is impossible to have S allow cc while it forbids c, since c is a marked prefix of cc. Indeed, this would require S being able to "unmark" some marked states of the plant, which it cannot do.

2.1.7 Maximal permissiveness and uniqueness

An important requirement in the basic supervisory control problem defined below (Section 2.1.8) is *maximal permissiveness*, namely, the fact that the supervisor must disable events only when strictly necessary to enforce the other requirements (non-blockingness or safety). This is a reasonable requirement, as it forces the supervisor to "disturb" the plant as little as possible, and only when strictly necessary. An important feature of the basic supervisory control framework is that a *unique* maximally-permissive supervisor always exists. As we shall see, this is not generally the case in the reactive synthesis framework. In this section, we establish this uniqueness property.

First, we define what it means for a supervisor to be more permissive than another supervisor. Consider a plant G and two supervisors S_1, S_2 for G. We say that S_1 *is no more permissive than* S_2 iff $S_1(\sigma) \subseteq S_2(\sigma)$ for any σ. We say that S_2 *is strictly more permissive than* S_1 iff S_1 is no more permissive than S_2 and $S_1 \neq S_2$.

Now, consider an admissible marked language L_{am} satisfying: (1) $L_{am} \subseteq \mathcal{L}_m(G)$ and (2) L_{am} is $\mathcal{L}_m(G)$-closed. A supervisor S which is non-blocking for G and safe w.r.t. L_{am} is said to be *maximally-permissive with respect to G and* L_{am} if there is no supervisor S' which is non-blocking for G, safe w.r.t. L_{am}, and strictly more permissive than S. Note that, a-priori, there could be more than one maximally-permissive supervisor, as the definition itself does not imply uniqueness. The theorem below shows that, for non-blockingness and safety, a unique maximally-permissive supervisor exists, provided that a supervisor exists at all.

Theorem 1 *Consider a plant G, and an admissible marked language L_{am} satisfying: (1) $L_{am} \subseteq \mathcal{L}_m(G)$ and (2) L_{am} is $\mathcal{L}_m(G)$-closed. If there exists a supervisor which is non-blocking for G and safe w.r.t. L_{am} then there exists a unique maximally-permissive supervisor S_{mpnb} which is non-blocking for G and safe w.r.t. L_{am}.*

Proof The books (Cassandras and Lafortune 2008; Wonham 2015) and the original papers (Ramadge and Wonham 1987; Wonham and Ramadge 1987) contain proofs of Theorem 1, as well as statements of necessary and sufficient conditions for the existence of a safe and non-blocking supervisor and algorithmic procedures for computing S_{mpnb}, given G and given an automaton representation of L_{am}. These proofs are normally done by defining the property of *controllability* of languages, showing that it is necessary and sufficient for the existence of a supervisor that exactly achieves a given language, and then proving the existence of the *supremal controllable sublanguage*.

For the reader interested in understanding the existence of a unique maximally permissive safe and non-blocking supervisor without reading more detailed treatments of supervisory control theory, we provide in Appendix A a direct proof based on disjunction of supervisors. This proof does not require the notions of controllable languages and of supremal controllable languages. □

In the sequel, the unique maximally-permissive non-blocking and safe supervisor will be denoted by S_{mpnb} and its associated closed-loop marked language by $\mathcal{L}_m(S_{mpnb}/G) = L_{am}^{mpnb}$. Since S_{mpnb} is non-blocking, then $\mathcal{L}(S_{mpnb}/G) = \overline{L_{am}^{mpnb}}$. Moreover, as a consequence of the maximal permissiveness property of S_{mpnb}, the language L_{am}^{mpnb} must contain the closed-loop marked language $\mathcal{L}_m(S_{other}/G)$ of any safe and non-blocking supervisor S_{other} for G wrt L_{am}.

As an example, consider again plant G_1 of Fig. 3, admissible marked language $L_2 = \{uc_2\}$, and supervisor S_3 defined above. S_3 is maximally-permissive w.r.t. G_1 and L_2. Indeed, any other supervisor, in order to be strictly more permissive than S_3, would have to either allow c_1 initially, which would violate safety w.r.t. L_2, or allow c_2 initially, which would violate non-blockingness, or allow c_1 after u, which again would be blocking.

Remark 4 (Non-uniqueness of supervisors achieving maximal behavior) Note that, although S_{mpnb} is unique, there are generally more than one supervisors that result in the same maximal closed-loop marked behavior L_{am}^{mpnb} since, by definition, S_{mpnb} might enable infeasible controllable events. As an example, consider the plant G_3 shown in Fig. 5, where both c_1, c_2 are controllable events. Note that all states of G are accepting and as a result, $\mathcal{L}_m(G)$ is prefix-closed and $\mathcal{L}_m(G) = \mathcal{L}(G) = \{\varepsilon, c_1, c_1c_2\}$.

Let $L_{am} := \{\varepsilon, c_1\}$, which can be interpreted as "c_2 should never occur". The maximally-permissive supervisor w.r.t. G_3 and L_{am} defined as above is

$$S_{mpnb}(\sigma) = \begin{cases} \{c_1\} & \text{if } \sigma = c_1 \\ \{c_1, c_2\} & \text{otherwise} \end{cases}$$

Another, less permissive supervisor, is one that always disables c_2. Both these two supervisors, however, achieve the same maximal closed-loop behavior, which is exactly L_{am}.

2.1.8 BSCP-NB: basic supervisory control problem with non-blockingness

We are now ready to define the standard supervisory control problem:

Definition 1 (BSCP-NB) Given DES G and admissible marked language $L_{am} \subseteq \mathcal{L}_m(G)$, with L_{am} assumed to be $\mathcal{L}_m(G)$-closed, find, if it exists, or state that there does not exist, a supervisor for G which is non-blocking for G, safe w.r.t. L_{am}, and maximally-permissive.

Observe that from the safety property $\mathcal{L}_m(S/G) \subseteq L_{am}$, we get $\overline{\mathcal{L}_m(S/G)} \subseteq \overline{L_{am}}$. Also, from non-blockingness we know that $\mathcal{L}(S/G) = \overline{\mathcal{L}_m(S/G)}$. These two properties imply $\mathcal{L}(S/G) \subseteq \overline{L_{am}}$ and thus in BSCP-NB the controlled behavior always stays within the prefix-closure of the admissible marked behavior.

As an example, consider again plant G_1 of Fig. 3, admissible marked language $L_2 = \{uc_2\}$, and supervisor S_3 defined above. S_3 is a solution to this BSCP-NB instance, since it is non-blocking, safe, and maximally-permissive, as explained above.

It may happen that BSCP-NB has no solution. For instance, suppose that $\mathcal{L}(G) = \overline{\{ucuc\}}$ and $\mathcal{L}_m(G) = \{uc, ucuc\}$ with $E_c = \{c\}$ and $E_{uc} = \{u\}$. Take $L_{am} = \{uc\}$. Then no safe

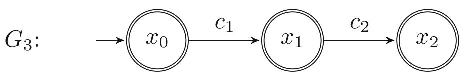


Fig. 5 Plant G_3

and non-blocking supervisor exists. Any supervisor will allow uncontrollable event u at the beginning of system operation. But enabling c after observing string u will violate safety, since string ucu will be in the closed-loop language. On the other hand, disabling c after observing string u causes deadlock. Hence, BSCP-NB has no solution in this example.

This shows that the set of uncontrollable events E_{uc} plays a central role in BSCP-NB. Algorithmic procedures that solve BSCP-NB must account for both uncontrollability and non-blockingness, and these two requirements are interdependent.

We postpone the discussion of algorithms to solve BSCP-NB to Section 3.4.1.

Example 2 (Coffee Machine Revisited) We revisit the coffee machine example at the beginning of Section 2.1, where the "plant" is the automaton G in Fig. 2. To obtain an instance of BSCP-NB, we formalize the specifications of safety and the two allowed recipes described earlier in the form of a language $L_{am} \subseteq \mathcal{L}_m(G)$. It is not hard to see that L_{am} is marked by the non-blocking automaton H shown in Fig. 6. This automaton ensures that grinding precedes brewing, that no grinding occurs after brewing has started, and it allows either one of the two recipes: gbb or $ggbbb$. Its marked language is also a sublanguage of $\mathcal{L}_m(G)$ as we have included self-loops for event c at the states where the coffee machine is not idle, consistent with the structure of G. Observe that automaton H needs to count the number of g and b events, something that is not done in G. It is also straightforward to verify from Fig. 6 that L_{am} satisfies the $\mathcal{L}_m(G)$-closure condition.

If $E_{uc} = \{c\}$ but all other events are controllable, then the solution of BSCP-NB achieves L_{am} exactly under control, i.e., the maximally permissive non-blocking supervisor S_{mpnb} is such that $\mathcal{L}_m(S_{mpnb}/G) = L_{am}$. In other words, in this simple example, the synthesis step is trivial since the specification language is exactly achievable by disabling g, b, or r at the right moment along each run of the plant. Hence, the closed-loop language $\mathcal{L}(S_{mpnb}/G) = \overline{L_{am}}$ is also equal to $\mathcal{L}(H)$. (This is of course not true in general.) Indeed, initially, $S_{mpnb}(\varepsilon) = \{c, b, g, r\}$: only c is feasible and it is uncontrollable, so it must be enabled; the other events are infeasible but added according to the definition of S_{mpnb}. Then the supervisor issues the following control actions for the given observed strings of G:
$S_{mpnb}(c) = \{c, g\}$ (i.e., b and r must be disabled to allow grinding to start);
$S_{mpnb}(cg) = \{c, g, b\}$ (i.e., r must be disabled until a recipe is completed);
$S_{mpnb}(cgb) = \{c, b\}$ (i.e., b must continue since the first recipe is being followed);

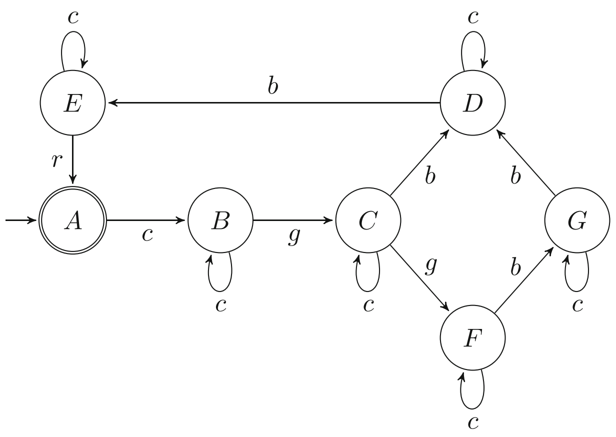


Fig. 6 Automaton H that marks the language L_{am} for the coffee machine example

$S_{mpnb}(cgg) = \{c, b\}$ (i.e., b must start since the second recipe is being followed);
$S_{mpnb}(cgbb) = \{c, r\}$ (i.e., r must start since the first recipe is completed);
$S_{mpnb}(cggb) = \{c, b\}$ (i.e., b must continue since the second recipe is being followed);
$S_{mpnb}(cggbb) = \{c, b\}$ (i.e., b must continue since the second recipe is being followed);
$S_{mpnb}(cggbbb) = \{c, r\}$ (i.e., r must start since the second recipe is completed);
$S_{mpnb}(cgbbr) = S_{mpnb}(cggbbbr) = \{c, b, g, r\}$ (i.e., a new cycle can begin);
and so forth.

If either event g, b, or r were uncontrollable, then BSCP-NB would have no solution. In the case where $g \in E_{uc}$ for instance, the strings cg^n, $n \geq 3$, which are in $\mathcal{L}(G)$, cannot be prevented by control, and they are outside $\overline{L_{am}}$. Similarly for string cb if b is uncontrollable, and for string cr if r is uncontrollable.

2.2 Reactive synthesis

In reactive synthesis, we build correct-by-construction controllers from declarative specifications. Controllers are *open dynamical systems*. A controller is open in the sense that it has inputs and outputs, and its behavior (its dynamics) depends on the inputs that the controller receives. These inputs come from the controller's *environment* (which may also be an open system, receiving as inputs the controller's outputs). A specification is declarative in the sense that it states how a controller must behave, but is not concerned with its internal structure. Rather, the specification only describes the desired behavior of the controller on the interface level, i.e., using its sets of inputs and outputs.

Let us illustrate the reactive synthesis framework by re-stating the coffee maker example (Section 2.1) in this framework. Consider the interface of the controller of a coffee maker that is depicted in Fig. 7.

The controller is meant to trigger the mechanical components of the coffee maker. The interface shows that we have one *input signal*, c. In this example, it is supposed to represent whether the user of the coffee maker has pressed the coffee button. There are also two *output signals*, namely b and g. While b is supposed to represent whether the brewing unit of the coffee maker is activated, g represents whether the grinding unit is activated.

In reactive synthesis, we assume that the controller evolves in steps, i.e., the controller communicates via input and output *signals*, which all have some value assigned in *every* time step. For simplicity, we assume Boolean signals where all values are Boolean. Controllers can therefore be viewed as state machines of type Moore or Mealy. In typical targets for reactive synthesis such as on-chip controllers, this assumption is well-justified, as there is typically a global clock generator in such systems. In the scope of our coffee maker, which serves mainly as an introductory example, we just choose a reasonable step duration.

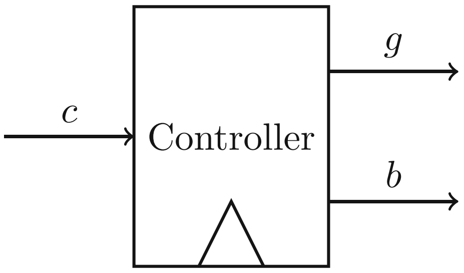


Fig. 7 The interface of a controller for a coffee maker

A reactive system has no designated time of going out-of-service, i.e., for every number of time steps n, we should not synthesize a controller that only works under the assumption that it runs for at most n steps, as letting it run for $n+1$ time steps is also conceivable. To abstract from this problem, we assume that the controller *never* goes out of service, and thus runs for an infinite duration. Such an execution produces a *trace*, which describes in which steps which inputs and outputs are set (i.e., have value **true**). Formally, a trace is an infinite word $w = w_0 w_1 w_2 \ldots$, where for every $i \in \mathbb{N}$, we have $w_i \subseteq \mathsf{AP}_I \cup \mathsf{AP}_O$, where AP_I is the set of input signals, and AP_O is the set of output signals. In the case of the coffee maker, $\mathsf{AP}_I = \{c\}$ and $\mathsf{AP}_O = \{g, b\}$. The following example shows an example trace of a coffee maker controller:

$$w = \begin{pmatrix} c \mapsto \textbf{false} \\ g \mapsto \textbf{false} \\ b \mapsto \textbf{false} \end{pmatrix} \begin{pmatrix} c \mapsto \textbf{true} \\ g \mapsto \textbf{true} \\ b \mapsto \textbf{false} \end{pmatrix} \begin{pmatrix} c \mapsto \textbf{false} \\ g \mapsto \textbf{true} \\ b \mapsto \textbf{false} \end{pmatrix} \begin{pmatrix} c \mapsto \textbf{false} \\ g \mapsto \textbf{false} \\ b \mapsto \textbf{true} \end{pmatrix} \begin{pmatrix} c \mapsto \textbf{false} \\ g \mapsto \textbf{false} \\ b \mapsto \textbf{true} \end{pmatrix} \begin{pmatrix} c \mapsto \textbf{false} \\ g \mapsto \textbf{false} \\ b \mapsto \textbf{true} \end{pmatrix} \ldots \quad (2)$$

In this trace, the coffee button is pressed in the second step, and grinding is performed in the two steps starting with the second one. Then, the brewing unit of the coffee maker is triggered for three steps.

This behavior of the controller could be one that satisfies its specification. For example, a specification for a coffee maker controller could be that once the coffee button is pressed, grinding should happen for two steps, and afterwards brewing should be done for three time steps while the grinding unit is idle.

To now perform synthesis from this specification, we need to formalize it. In reactive synthesis, this is typically done by describing the specification in a *logic*. The logic *CTL** (Emerson and Halpern 1986) is well-suited for this purpose and extends standard Boolean logic by *temporal operators* and *path quantifiers* that intuitively allow us to connect the system's signal valuations in one step with the actions in other, future time steps. In the context of logic, we also call the signals *atomic propositions*. (CTL stands for Computation Tree Logic.) The informal specification from the previous paragraph would be formalized into CTL* as follows:

$$\psi = \mathsf{AG}\,(c \rightarrow (g \wedge \mathsf{X}g \wedge \mathsf{XX}((b \wedge \neg g) \wedge \mathsf{X}(b \wedge \neg g) \wedge \mathsf{XX}(b \wedge \neg g))))$$

The formula starts with the path quantifier A, which denotes that the expression right of the operator should hold along all executions of a system to be synthesized. It is followed by the temporal operator G, which is called the *globally* operator. For some formula $\mathsf{G}\phi$ to hold at some point in the execution of a system, ϕ needs to hold for all steps from that point onward.

A specification is required to hold right from the start of the system. Thus, prefixing our coffee maker specification ψ with AG means that the implication $c \rightarrow \ldots$ has to hold at every step of the system's execution. The implication in turn describes that $(g \wedge \mathsf{X}g \wedge \ldots)$ shall happen whenever we have c, i.e., the coffee button has been pressed. The consequent of the implication now is $(g \wedge \mathsf{X}g \wedge \mathsf{XX}((b \wedge \neg g) \wedge \mathsf{X}(b \wedge \neg g) \wedge \mathsf{XX}(b \wedge \neg g))$, which is a Boolean formula in which the temporal operator X (*next*) is used. It describes that we need to look one step into the future in the trace of the system to test if some sub-formula holds. So $\mathsf{X}g$ holds in the first step in a trace of the system if g holds in the second step of the trace. Likewise, $\mathsf{XX}g$ holds in the first step in a trace if g holds in the third step of the trace. This example also shows that the operators in CTL* can be chained, which makes it a rich modeling formalism for specifications. Note that the consequent of the implication in ψ describes the informal statement of what shall happen upon a coffee button press from above in a formal way.

To actually synthesize a system from a specification, the specification needs to be *realizable*, i.e., there has to exist a system implementation for the given interface that ensures that every trace of the system satisfies the specification. Most synthesis algorithms also check realizability, i.e., they do not only synthesize an implementation for realizable specifications, but also detect unrealizability. As a consequence, there is often no distinction between the two steps in the literature.

While testing realizability appears to be trivial and unnecessary, in the practice of synthesis, it is not. For example, the coffee maker specification from above is unrealizable, and despite its short length, this fact is easily overlooked. The reason for unrealizability here is that we might press the coffee button in the first two successive steps of the system's execution. The specification part $\mathsf{X}g$ then requires that grinding is performed in the third step (as the implication is triggered by the second button press), but at the same time the specification part $\mathsf{XX}((b \wedge \neg g) \wedge \ldots)$ requires that grinding does not happen in the third step. This is a contradiction that the system to be synthesized cannot avoid, as the input is not under its control. Therefore, this specification is unrealizable.

There are two ways to fix the specification. One is to allow the system to delay the production of the next cup until a grinding and brewing cycle has finished. This can be done using the *eventually operator* (F) of CTL*. Intuitively, a CTL* formula $\mathsf{F}\phi$ holds at a point in a trace of the system if at some point in the future, ϕ holds. The modified specification then looks as follows:

$$\psi = \mathsf{AG}\,(c \rightarrow \mathsf{F}(g \wedge \mathsf{X}g \wedge \mathsf{XX}((b \wedge \neg g) \wedge \mathsf{X}(b \wedge \neg g) \wedge \mathsf{XX}(b \wedge \neg g)))$$

It is now realizable, partly because there is no requirement that the number of coffees produced matches the number of button presses. Note that the eventually operator does not impose a bound on the number of steps by which a brewing cycle might be delayed. Thus, a system that satisfies this specification could react with a delay that gets longer and longer the more coffees are made. However, none of the contemporary synthesis algorithms produces such implementations, as such a behavior would require an infinite-state implementation, but they only compute finite-state ones. As it can be shown that whenever there exists an implementation for a CTL* specification, there also exists a finite-state one, this is also not necessary. So using the eventually operator in this context instead of imposing a maximal bound on the number of steps until when grinding should start is reasonable.

Another possibility to fix the specification is to add an *assumption* to the specification that expresses that the coffee button cannot be pressed when brewing or grinding is already happening. The new realizable specification would be:

$$\psi' = \mathsf{A}\,(\mathsf{G}((g \vee b) \rightarrow \neg\mathsf{X}c) \rightarrow \mathsf{G}\,(c \rightarrow \mathsf{F}(\mathsf{X}g \wedge \mathsf{XX}g \wedge \mathsf{XXX}((b \wedge \neg g) \wedge \mathsf{X}(b \wedge \neg g) \wedge \mathsf{XX}(b \wedge \neg g)))))$$

An assumption of course always has to be reasonable in practice to make sense in synthesis. If we know that the coffee maker in which the controller is supposed to work ensures that the button cannot be pressed while the maker is running (or alternatively ignores the button press), then the assumption is justified.

After this short introduction to the aims of reactive synthesis, let us now discuss more formally how we specify the intended behavior of the system to be synthesized and how such a system is actually represented.

2.2.1 *Computation trees*

A reactive system has to satisfy a specification regardless of the input to the system. To get an overview about the possible behaviors of a system, for the scope of synthesis, we typically view a system implementation as a *computation tree*, as these describe all system behaviors of a reactive system at once. Formally, for some interface $(\mathsf{AP}_I, \mathsf{AP}_O)$ of a reactive system, a computation tree is a tuple $\langle T, \tau\rangle$, where $T = (2^{\mathsf{AP}_I})^*$ and $\tau : T \rightarrow 2^{\mathsf{AP}_I \cup \mathsf{AP}_O}$. The tree describes all the possible traces by having τ map every input sequence to the system to an output signal valuation that the system produces after having read the input sequence. Without loss of generality, we assume that every node in the computation tree is also labeled by the last input, i.e., we have $\tau(t_0 \ldots t_n)|_{\mathsf{AP}_I} = t_n$ for every $t_0 \ldots t_n \in (2^{\mathsf{AP}_I})^+$. While labeling the nodes in the tree according to the last direction seems to be unnecessary, it allows us to define the logic CTL* below in a way that generalizes to applying the logic to Kripke structures (which we define in Section 3.2.1) as well. Note that $\tau(\epsilon)|_{\mathsf{AP}_I}$ is not constrained in any way and can be freely set by the computation tree.

Figure 8 shows an example computation tree of a coffee maker controller.

2.2.2 *The temporal logic CTL**

Let AP be a set of *atomic propositions*. Expressions in CTL* can either be state formulas or path formulas. We define the set of path formulas in the temporal logic CTL* inductively by the following rules:

- every CTL* state formula is also a CTL* path formula
- For every CTL* path formula ψ, we have that $\neg\psi$, $\mathsf{G}\psi$, $\mathsf{F}\psi$, and $\mathsf{X}\psi$ are also CTL* path formulas;
- For all CTL* path formulas ψ and ψ', we have that $\psi\,\mathsf{U}\psi'$, $\psi\,\mathsf{R}\psi'$, $\psi \vee \psi'$, and $\psi \wedge \psi'$ are also CTL* path formulas.

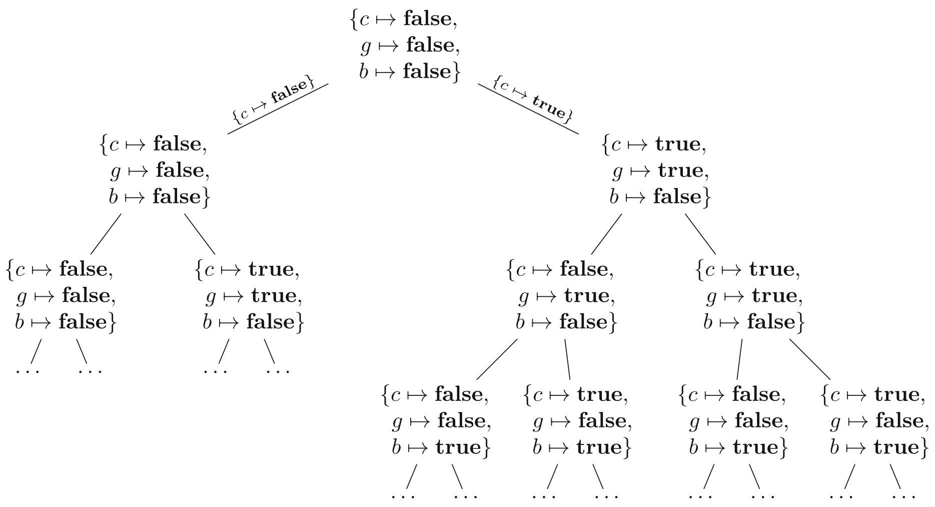

{c ↦ false, g ↦ false, b ↦ false}
{c ↦ false}
{c ↦ true}
{c ↦ false, g ↦ false, b ↦ false}
{c ↦ true, g ↦ true, b ↦ false}
{c ↦ false, g ↦ false, b ↦ false}
{c ↦ true, g ↦ true, b ↦ false}
{c ↦ false, g ↦ true, b ↦ false}
{c ↦ true, g ↦ true, b ↦ false}
{c ↦ false, g ↦ false, b ↦ true}
{c ↦ true, g ↦ false, b ↦ true}
{c ↦ false, g ↦ false, b ↦ true}
{c ↦ true, g ↦ false, b ↦ true}
...

Fig. 8 A computation tree of a coffee maker controller. Taking a branch to the left always refers to the input $\{c \mapsto \textbf{false}\}$, whereas the right branches always refer to the input $\{c \mapsto \textbf{true}\}$

The set of state formulas is defined as follows:

- For every $p \in \mathsf{AP}$, p is a CTL* state formula;
- For all CTL* state formulas ϕ and ϕ', we have that $\phi \vee \phi'$, $\phi \wedge \phi'$ and $\neg\phi'$ are also CTL* state formulas;
- Given a CTL* path formula ψ, $\mathsf{A}\psi$ and $\mathsf{E}\psi$ are CTL* state formulas.

The semantics of CTL* is defined over computation trees. Let AP be a set of atomic propositions for $\mathsf{AP} = \mathsf{AP}_I \cup \mathsf{AP}_O$, ψ be a CTL* formula over AP, and $\langle T, \tau\rangle$ be a computation tree. A *branch* in $\langle T, \tau\rangle$ starting in some node $t \in T$ is defined to be a sequence $b = b_0 b_1 b_2 \ldots$ such that (1) $b_0 = t$, (2) for every $i \in \mathbb{N}$, we have $b_i \in T$, and (3) for every $i \in \mathbb{N}$, we have $b_{i+1} = b_i x$ for some $x \subseteq \mathsf{AP}_I$. We denote the suffix of a string b starting at position $j \in \mathbb{N}$ by b^j, i.e., for $b = b_0 b_1 b_2 \ldots$, we have $b^j = b_j b_{j+1} b_{j+2} \ldots$.

Given some node $t = t_0 \ldots t_n$ of $\langle T, \tau\rangle$, we evaluate the validity of a CTL* state formula at point t by recursing over the structure of the CTL* state formula (where ψ is a CTL* path formula and ϕ and ϕ' are CTL* state formulas):

- $\langle T, \tau\rangle, t \models p$ for some $p \in \mathsf{AP}_I \cup \mathsf{AP}_O$ if $p \in \tau(t)$;
- $\langle T, \tau\rangle, t \models \neg\phi$ if and only if not $\langle T, \tau\rangle, t \models \phi$;
- $\langle T, \tau\rangle, t \models \phi \vee \phi'$ if and only if $\langle T, \tau\rangle, t \models \phi$ or $\langle T, \tau\rangle, t \models \phi'$;
- $\langle T, \tau\rangle, t \models \phi \wedge \phi'$ if and only if $\langle T, \tau\rangle, t \models \phi$ and $\langle T, \tau\rangle, t \models \phi'$;
- $\langle T, \tau\rangle, t \models \mathsf{A}\psi$ if for all branches b starting from t, we have $\langle T, \tau\rangle, b \models \psi$;
- $\langle T, \tau\rangle, t \models \mathsf{E}\psi$ if for some branch b starting from t, we have $\langle T, \tau\rangle, b \models \psi$.

Likewise, given some branch $b = b_0 b_1 \ldots$ of $\langle T, \tau\rangle$, we evaluate the validity of a CTL* path formula on b by recursing over the structure of the CTL* path formula (where ψ and ψ' are CTL* path formulas and ϕ is a CTL* state formula):

- $\langle T, \tau\rangle, b \models \phi$ if and only if $\langle T, \tau\rangle, b_0 \models \phi$;
- $\langle T, \tau\rangle, b \models \neg\psi$ if and only if not $\langle T, \tau\rangle, b \models \psi$;
- $\langle T, \tau\rangle, b \models \psi \vee \psi'$ if and only if $\langle T, \tau\rangle, b \models \psi$ or $\langle T, \tau\rangle, b \models \psi'$;
- $\langle T, \tau\rangle, b \models \psi \wedge \psi'$ if and only if $\langle T, \tau\rangle, b \models \psi$ and $\langle T, \tau\rangle, b \models \psi'$;
- $\langle T, \tau\rangle, b \models \mathsf{X}\psi$ if and only if $\langle T, \tau\rangle, b^1 \ldots \models \psi$;
- $\langle T, \tau\rangle, b \models \mathsf{G}\psi$ if and only if for all $j \in \mathbb{N}$, we have $\langle T, \tau\rangle, b^j \models \psi$;
- $\langle T, \tau\rangle, b \models \mathsf{F}\psi$ if and only if for some $j \in \mathbb{N}$, we have $\langle T, \tau\rangle, b^j \models \psi$;
- $\langle T, \tau\rangle, b \models \psi\,\mathsf{U}\psi'$ if and only if for some $j \in \mathbb{N}$, we have $\langle T, \tau\rangle, b^j \models \psi'$, and for all $0 \leq i < j$, we have $\langle T, \tau\rangle, b^i \models \psi$;
- $\langle T, \tau\rangle, b \models \psi\,\mathsf{R}\psi'$ if either for all $j \in \mathbb{N}$, we have $\langle T, \tau\rangle, b^j \models \psi'$, or there exists some $j \in \mathbb{N}$ such that $\langle T, \tau\rangle, b^j \models \psi$, and for all $i \leq j$, we have $\langle T, \tau\rangle, b^i \models \psi'$.

We define the set of trees for which all children of the root node satisfy some CTL* state formula ϕ to be the *models* of ϕ.[1]

[1]This definition differs a bit from the classical definition of a computation tree's containment in the tree language induced by a CTL* formula, where the safisfaction of ϕ is checked at the root node itself. For reactive synthesis, where every node is labeled by the last input, this definition is a bit awkward, however, as the root node does not have a last input. By evaluating a CTL* formula on all children of the root node, we circumvent this problem. For the scope of this paper, we find this approach to be conceptually cleaner than allowing the system to choose an arbitrary first input proposition valuation at the root, as done in many classical publications on reactive synthesis.

Given some CTL* state formula ϕ, we say that ϕ is realizable for some interface $\mathcal{I} = (\mathsf{AP}_I, \mathsf{AP}_O)$ if there exists an $2^{\mathsf{AP}_O \cup \mathsf{AP}_I}$-labeled 2^{AP_I}-tree that is a model of ϕ (and that copies the last input correctly to its node labels).

CTL is the subset of CTL* obtained by restricting the path formulas to be $\mathsf{X}\phi$, $\mathsf{F}\phi$, $\mathsf{G}\phi$, and $\phi \, \mathsf{U} \phi'$, where ϕ, ϕ' are CTL state formulas. LTL is the subset of CTL* consisting of the formulas $\mathsf{A}\phi$ in which the only state subformulas in ϕ are atomic propositions.

Definition 2 (Realizability Problem) Given some system interface $\mathcal{I} = (\mathsf{AP}_I, \mathsf{AP}_O)$ and some CTL* state formula ϕ (the specification), the realizability problem is to test if there exists some computation tree $\langle T, \tau \rangle$ with $T = (2^{\mathsf{AP}_I})^*$ and $\tau : T \to 2^{\mathsf{AP}_I \cup \mathsf{AP}_O}$ that copies the respective last input to its node labels correctly and such that $\langle T, \tau \rangle$ is a model of ϕ.

2.2.3 *Transducers*

The definition of the realizability problem above has one slight problem: while it clearly defines what constitutes a computation tree that represents a solution to the synthesis problem, such computation trees have infinitely many nodes. Thus, the model is not directly usable for actually synthesizing systems, which have to be finite-state in order to be implementable in the field. As a remedy to this problem, we define (deterministic) transducers here, which serve as finite generators for computation trees. It can be shown that for every realizable specification, there exists a computation tree that is generated by a transducer, and thus for the scope of synthesis, it suffices to search for a transducer that generates a suitable computation tree.

Formally, a *transducer* over some set of input atomic propositions AP_I and output atomic propositions AP_O is defined as a tuple $\mathcal{T} = (S, 2^{\mathsf{AP}_I}, 2^{\mathsf{AP}_I \cup \mathsf{AP}_O}, \delta, s_0, L)$, where S is a (finite) set of states, $\delta : S \times 2^{\mathsf{AP}_I} \to S$ is the transition function, $s_0 \in S$ is the initial state of the system, and $L : S \to 2^{\mathsf{AP}_I \cup \mathsf{AP}_O}$ assigns to each state its labeling. We require that the states always represent the last input to the transducer, i.e., we have $L(s)|_{\mathsf{AP}_I} = x$ for every $s \in S$ such that for some $s' \in S$, we have $\delta(s', x) = s$. The definition of a transducer corresponds to the definition of a *Mealy machine* that is common in the practice of hardware design, but with the addition that the transducer always produces the last output.

We say that some word $w = w_0 w_1 \ldots \in (2^{\mathsf{AP}_O} \times 2^{\mathsf{AP}_I})^\omega$ is a trace of $\mathcal{T}$ if there exists some sequence of states $\pi = \pi_0 \pi_1 \ldots \in S^\omega$ such that $\pi_0 = s_0$, and for all $i \in \mathbb{N}$, we have $\pi_{i+1} = \delta(\pi_i, x)$ for some $x \subseteq \mathsf{AP}_I$ and $w_i = L(\pi_i)$. We call π a *run* of the transducer in this context. We can obtain a computation tree $\langle T, \tau \rangle$ from a transducer $\mathcal{T} = (S, 2^{\mathsf{AP}_I}, 2^{\mathsf{AP}_I \cup \mathsf{AP}_O}, \delta, s_0, L)$ by setting $T = (2^{\mathsf{AP}_I})^*$ and $\tau(t_0 t_1 \ldots t_n) = L(\delta(\ldots \delta(\delta(s_0, t_0), t_1), \ldots, t_n))$ for all $t_0 t_1 \ldots t_n \in T$.

To illustrate the concept of transducers, Fig. 9 shows an example transducer for a coffee maker controller that could have the same set of traces as the computation tree in Fig. 8. As the computation tree in Fig. 8 is not fully shown (after all, it is infinite), we can however not be sure about that.

2.2.4 *Reactive Synthesis Problem (RSP)*

Definition 3 (RSP) Given some system interface $\mathcal{I} = (\mathsf{AP}_I, \mathsf{AP}_O)$ and some CTL* state formula ϕ (the specification), the reactive synthesis problem (RSP) for $\mathcal{I}$ and ϕ is to compute a transducer over $\mathcal{I}$ whose computation tree satisfies ϕ whenever it exists, and to deduce that no such transducer exists whenever this is the case.

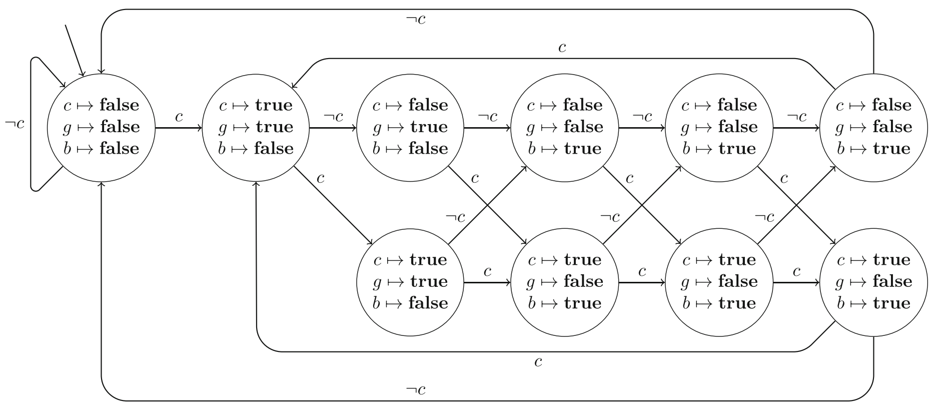


Fig. 9 An example transducer structure for a coffee maker controller with $\mathsf{AP}_I = \{c\}$ and $\mathsf{AP}_O = \{b, g\}$. The initial state is marked with an incoming arrow. Edges are labeled by simple Boolean formulas that represent the conditions over the input characters under which the transition is taken

We postpone a discussion of algorithms to solve RSP to Section 3.4.2.

2.2.5 *Maximal permissiveness in RSP*

The reader may have noted that in the definition of the supervisory control problem, we are concerned with computing maximally permissive controllers, but in the reactive synthesis problem, we just search for *any* controller that satisfies the specification. There are two reasons for this difference. First of all, the reactive synthesis problem was originally defined in this form by Church (1963). The second, more important reason is however that in general, maximally-permissive controllers do not exist.

To actually discuss maximally-permissiveness in the context of the reactive synthesis problem, we first of all need to change our transducer definition, as the transducers currently dealt with are deterministic. For a transducer $\mathcal{T} = (S, 2^{\mathsf{AP}_I}, 2^{\mathsf{AP}_O}, \delta, s_0, L)$, we redefine δ to map from $S \times 2^{\mathsf{AP}_I}$ to a subset of S. This way, whenever the controller is in some state s and reads some input $x \subseteq \mathsf{AP}_I$, then it can transition to any of the states in $\delta(s, x)$. We require that for all $s \in S$ and $x \subseteq \mathsf{AP}_I$, $\delta(s, x)$ is non-empty. We furthermore allow more than one initial state and modify the definition of a computation tree of account for these facts.

Computation trees for such non-deterministic transducers $\mathcal{T} = (S, 2^{\mathsf{AP}_I}, 2^{\mathsf{AP}_I \cup \mathsf{AP}_O}, \delta, S_0, L)$ are then tuples $\langle T, \tau \rangle$ with $T \subseteq S^*$ such that for some $s_0 \in S_0$, we have:

1.1. $\tau(\epsilon) = L(s_0)$
1.2. $|\{s \in S \mid s \in T\}| = |2^{\mathsf{AP}_I}|$ and $\{L(s)|_{2^{\mathsf{AP}_I}} : s \in S \wedge s \in T\} = 2^{\mathsf{AP}_I}$
1.3. For all $s \in S$ with $s \in T$, we have $s \in \delta(s_0, L(s)|_{2^{\mathsf{AP}_I}})$
2.1. For all $t = t_0 \ldots t_n \in T \setminus \{\epsilon\}$, we have $\tau(t_0 \ldots t_n) = L(t_n)$
2.2. For all $t = t_0 \ldots t_n \in T \setminus \{\epsilon\}$, we have that $|\{s \in S \mid ts \in T\}| = |2^{\mathsf{AP}_I}|$ and $\{L(ts)|_{2^{\mathsf{AP}_I}} : ts \in T\} = 2^{\mathsf{AP}_I}$
2.3. For all $t = t_0 \ldots t_n \in T \setminus \{\epsilon\}$ with $n \geq 1$, we have $t_n \in \delta(t_{n-1}, L(t_n)|_{2^{\mathsf{AP}_I}})$.

Note that we actually only have three different conditions, but for the sake of completeness need one copy of each condition for the root and one copy for the other nodes. The conditions together ensure that all the possible computation trees of a transducer are *input-complete*, i.e., from every node, they have one possible successor for every next input. We say that a non-deterministic transducer satisfies some CTL* state formula ϕ if every input-complete computation tree induced by the transducer satisfies ϕ (at the root).

We furthermore say that two computation trees $\langle T, \tau \rangle$ and $\langle T', \tau' \rangle$ are *isomorphic* if there exists some bijective function $f : T \to T'$ with $|f(t)| = |t|$ for all $t \in T$ and $\tau(t) = \tau'(f'(t))$ for all $t \in T$. Isomorphic trees effectively represent the same behavior of a reactive system although the internal structure of the transducers from which the trees are possibly generated may be different.

We call a non-deterministic transducer *maximally permissive* for some CTL* state formula specification ϕ and interface $\mathcal{I} = (\mathsf{AP}_I, \mathsf{AP}_O)$ if (1) the transducer branches over AP_I and satisfies ϕ on all trees induced by the transducer, and (2) every input-responsive computation tree for $\mathcal{I}$ that satisfies ϕ (at the root) has an isomorphic computation tree that is induced by the transducer.

Note that maximally permissive **finite-state** controllers/transducers do not exist in general for RSP. For example, let $\mathsf{AP}_I = \{r\}$, $\mathsf{AP}_O = \{g\}$, $\phi = \mathsf{AGF}g$, and $\mathcal{T} = (S, 2^{\mathsf{AP}_I}, 2^{\mathsf{AP}_I \cup \mathsf{AP}_O}, \delta, s_0, L)$ be *any* transducer that satisfies ϕ. Since $\mathcal{T}$ satisfies ϕ, there has to be some upper bound $b \in \mathbb{N}$ on the number of steps until g is set to true by the controller for the first time, as otherwise, there exists some path in some computation tree induced by $\mathcal{T}$ on which $\mathsf{GF}g$ is not satisfied. However, since a controller that sets g to true every $(b + 1)$th cycle satisfies ϕ as well, $\mathcal{T}$ cannot be maximally permissive. As we started with an arbitrary finite-state transducer, this proves that no controller can be maximally permissive.

Section 3.2.5 contains a more detailed discussion of maximal permissiveness in the reactive synthesis context.

3 Bridging the gap

In this section, we discuss how to bridge the gap between the synthesis problems considered in supervisory control and reactive synthesis. We do this by establishing relations (e.g., reductions) between some specific problems studied in these frameworks. The general situation is described in Fig. 10, where the basic supervisory control problem (BSCP-NB) and the reactive synthesis problem (RSP) are the cases at the cliffs of the gap. We introduce problems that conceptually lie in between BSCP-NB and RSP in order to bridge the gap. These problems always differ in one aspect from their neighbors, and we can perform reductions between these problems. As a result, we can move gently between the BSCP-NB and RSP problems, which simplifies understanding the concepts to follow.

However, our bridge does not exactly meet in the middle. The reason is that the aim of supervisory control and the aim of reactive synthesis slightly differ. In supervisory control, we always want our supervisor to be maximally permissive (being a "parent"), as it should only block undesired actions. In reactive synthesis, on the other hand, where maximal permissiveness is unachievable in general, we want our controller to actively enforce certain properties, possibly at the expense of preventing certain overall system behavior that is

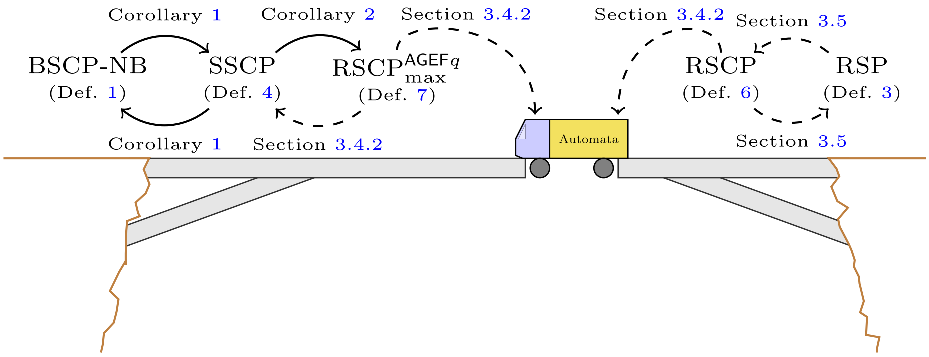


Fig. 10 Relations between different synthesis and control problems

unproblematic. This mismatch, and the lack of study of the general reactive synthesis problem with maximal permissiveness, RSCP$_{\max}$ (see Definition 8 that follows), prevent us from performing a sequence of reductions that map the problems completely onto each other.

Our focus in this paper is on the left-side of the bridge: we present formal reductions from BSCP-NB to a special instance of RSCP$_{\max}$. Specifically, we show in Section 3.1 that BSCP-NB is equivalent to a simpler supervisory control problem in which only non-blockingness is required, called SSCP, and which simplifies the reduction to the reactive synthesis setting. In Section 3.2 we define a reactive synthesis problem with an explicit notion of plants, called RSCP. This makes it easier to capture supervisory control problems where the plant is an input to the problem. RSCP does not generally admit maximally-permissive solutions, but does so for the non-blocking requirement, which is the only requirement of SSCP. In Section 3.3 we show how SSCP can be reduced to a variant of RSCP which requires maximal permissiveness.

Regarding the right side of the bridge, we discuss in Section 3.5 the links between reactive synthesis with plants and reactive synthesis without plants.

The center of the bridge is represented as an "automaton vehicle." This vehicle captures the fact that at an algorithmic level, the problems on either side of the gap are solved by applying automata-theoretic techniques, as discussed in Section 3.4.2.

3.1 Simplifying the supervisory control problem

In view of reducing BSCP-NB to the reactive synthesis framework, we first reduce BSCP-NB to a simpler problem. In particular, we will eliminate the safety specification L_{am} by incorporating it into the plant. This can be done by taking as new plant the product of the original plant G and an automaton recognizing L_{am}. The simpler problem asks for a non-blocking supervisor for the new plant. We next formalize this idea.

3.1.1 Incorporating safety into the plant

Let $G = (X, x_0, X_m, E, \delta)$ be a DES plant. Let $L_{am} \subseteq \mathcal{L}_m(G)$ and let L_{am} be $\mathcal{L}_m(G)$-closed. Let $A = (X^A, x_0^A, X_m^A, E, \delta^A)$ be a deterministic finite-state automaton such that

$\mathcal{L}_m(A) = L_{am}$ and $\mathcal{L}(A) = E^*$. We can assume without loss of generality that A is complete in the sense that its transition function δ^A is total. Therefore, every string in E^* has a unique run in A, although only some runs will generally end up in a marked state; in fact, A is generally a blocking automaton, since strings outside $\overline{L_{am}}$ will never reach a marked state. (The fact that $\mathcal{L}(A) \nsubseteq \mathcal{L}(G)$ is not a problem; this will become apparent below.)

The *product of G and A*, denoted by $G \times A$, is defined to be the automaton

$$G \times A = (X \times X^A, (x_0, x_0^A), X_m \times X_m^A, E, \delta')$$

such that

$$\delta'\Big((x, x^A), e\Big) = \begin{cases} \big(\delta(x, e), \delta^A(x^A, e)\big) \\ \text{if } \delta(x, e) \text{ is defined } (\delta^A \text{ is total, so } \delta^A(x^A, e) \text{ is always defined}) \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

It follows from the construction of A and the assumptions on L_{am} and A that

$$\begin{aligned} \mathcal{L}(G \times A) &= \mathcal{L}(G) \cap \mathcal{L}(A) = \mathcal{L}(G) \cap E^* = \mathcal{L}(G) \quad \text{and} \\ \mathcal{L}_m(G \times A) &= \mathcal{L}_m(G) \cap \mathcal{L}_m(A) = \mathcal{L}_m(G) \cap L_{am} = L_{am}. \end{aligned}$$

G and $G \times A$ have the same set of events E, thus also the same subsets of controllable and uncontrollable events. Therefore, any supervisor S for G is also a supervisor for $G \times A$, and vice versa. This allows us to state the following result.

Theorem 2 *Let $G = (X, x_0, X_m, E, \delta)$, $L_{am} \subseteq \mathcal{L}_m(G)$, and assume that L_{am} is $\mathcal{L}_m(G)$-closed. Let A be a complete DFA such that $\mathcal{L}_m(A) = L_{am}$. Let S be a supervisor for G, and therefore also for $G \times A$. Then, the following statements are equivalent:*

1. *S solves BSCP-NB for plant G with respect to admissible marked language L_{am}.*
2. *S solves BSCP-NB for plant $G \times A$ with respect to admissible marked language $\mathcal{L}_m(G \times A)$.*

Proof The proof is given in Appendix B. □

To the best of our knowledge, the result in Theorem 2 does not appear explicitly in the published literature, although it can be inferred rather straightforwardly from the algorithmic procedure for solving BSCP-NB originally presented in Ramadge and Wonham (1987), Wonham and Ramadge (1987). Our proof in Appendix B is direct and relies only on the concepts introduced so far in this paper.

3.1.2 SSCP: simple supervisory control problem

Theorem 2 allows to reduce BSCP-NB to a simpler problem, namely, that of finding a maximally-permissive non-blocking supervisor for a given plant, *with no external admissible marked behavior.* We call the resulting problem the *Simple Supervisory Control Problem* (SSCP) and restate it formally:

Definition 4 (SSCP) Given DES G, find (if it exists, or state that none exists) a maximally-permissive non-blocking supervisor for G, that is, a supervisor S which is non-blocking for G, and such that there is no supervisor S' which is non-blocking for G and strictly more permissive than S.

Corollary 1 *BSCP-NB and SSCP are equivalent problems, i.e., each one can be reduced to the other with a polynomial-time reduction.*

Proof SSCP is equivalent to the special case of BSCP-NB with $L_{am} := \mathcal{L}_m(G)$. This is because $\mathcal{L}_m(G)$ is trivially $\mathcal{L}_m(G)$-closed and $\mathcal{L}_m(S/G) \subseteq \mathcal{L}_m(G)$ always holds. Obviously this special case of BSCP-NB can be reduced to BSCP-NB. Conversely, Theorem 2 demonstrates that BSCP-NB can be reduced to this special case of BSCP-NB. This reduction is polynomial-time because $G \times A$ can be computed in polynomial time from G and A. Therefore all three problems, BSCP-NB, BSCP-NB with $L_{am} := \mathcal{L}_m(G)$, and SSCP are equivalent with polynomial-time reductions. □

It also follows from the above results and Theorem 1 that if a solution to SSCP exists, then this solution is unique, i.e., the maximally-permissive non-blocking supervisor is unique.

3.1.3 Finite-memory, state-based supervisors

We will use SSCP to establish a precise connection between supervisory control and reactive synthesis. In this regard, we prove a useful property for the type of supervisors that need to be considered in solving SSCP.

A supervisor is a function $S : E^* \to 2^E$. The domain of this function is E^*, which is an infinite set. This makes it *a priori* possible for S to require infinite memory. Fortunately, it can be shown that finite-memory, and in particular *state-based* supervisors, are sufficient for SSCP.

Definition 5 Let $G = (X, x_0, X_m, E, \delta)$ be a DES plant with $E = E_c \cup E_{uc}$ and let $S : E^* \to 2^E$ be a supervisor for G. S is said to be *state-based* if

$$\forall \sigma_1, \sigma_2 \in E^* : \delta(x_0, \sigma_1) = \delta(x_0, \sigma_2) \implies S(\sigma_1) = S(\sigma_2).$$

That is, S is state-based if it outputs the same decision for two behaviors σ_1, σ_2 of G that end up in the same state. Therefore, S only needs to know the current state of the plant in order to decide which controllable events should be allowed. Note that we assume that, when one or both of $\delta(x_0, \sigma_1), \delta(x_0, \sigma_2)$ are undefined, the equality $\delta(x_0, \sigma_1) = \delta(x_0, \sigma_2)$ is false, and therefore in that case the implication is trivially true. Given that, Definition 5 does not constrain the structure of S outside of $\mathcal{L}(G)$, i.e., for strings in $E^* \setminus \mathcal{L}(G)$. This is because we do not need to make any assumptions regarding the form of S over $E^* \setminus \mathcal{L}(G)$, since these control actions will never be invoked when S is applied to G.

Let S be a state-based supervisor for G, and assume all states of G are *reachable*, that is, $\forall x \in X : \exists \sigma \in E^* : \delta(x_0, \sigma) = x$; note that unreachable states can be removed from X without affecting SSCP. Then the action of S on G can be viewed as a function $S' : X \to 2^E$, where $S'(x) = S(\sigma)$ where σ is any string such that $\delta(x_0, \sigma) = x$ (x is reachable, so at least one such σ exists). Because S is state-based, it returns the same choice $S(\sigma') = S(\sigma)$ for any other string σ' such that $\delta(x_0, \sigma') = x$. Therefore, S' is well-defined. Thus, we can assume, without loss of generality, that a state-based supervisor for G is a function $S : X \to 2^E$. As in the case of general supervisors, we assume that $E_{uc} \subseteq S(x)$, for all $x \in X$, to ensure that a state-based supervisor never disables an uncontrollable event.

In addition, the definition of the closed-loop system S/G can be simplified in the case of state-based supervisors. In particular, S/G can be defined as $S/G = (X', x_0', X_m', E, \delta')$

where $X' = X$ (instead of $X \times L(G)$), $x'_0 = x_0$, $X'_m = X_m$, and

$$\delta'(x, e) = \begin{cases} \delta(x, e) & \text{if } \delta(x, e) \text{ is defined and } e \in S(x) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The following result states that in order to solve the SSCP it suffices to consider only state-based supervisors.

Theorem 3 *The solution to SSCP, if it exists, is a state-based supervisor.*

Proof Let S_{mpnb} be the unique solution of SSCP. Suppose that $\delta(x_0, \sigma_1) = \delta(x_0, \sigma_2) = x_p$ but that $S_{mpnb}(\sigma_1) \neq S_{mpnb}(\sigma_2)$. Then there must exist $e_c \in E_c$ such that $e_c \in S_{mpnb}(\sigma_1) \setminus S_{mpnb}(\sigma_2)$. (Recall that we assume that all uncontrollable events are always enabled by a supervisor.) Since $\delta(x_0, \sigma_1) = \delta(x_0, \sigma_2) = x_p$, the post-languages in G from x_p are the same, i.e.,

$$\begin{aligned} \mathcal{L}(G)/\sigma_1 &= \mathcal{L}(G)/\sigma_2 \\ \mathcal{L}_m(G)/\sigma_1 &= \mathcal{L}_m(G)/\sigma_2 \end{aligned}$$

Since in SSCP the control problem involves the simple safety specification $\mathcal{L}_m(G)$ and the non-blocking property, as captured by the marked states of G, the decision to enable or not an event after a given (safe) string σ depends entirely on the post-language after σ, equivalently, on the state $\delta(x_0, \sigma)$ reached by σ, since the states of G are equivalence classes for future behavior. (Recall Lemma 1.) Thus, no such e_c can exist at state x_p, otherwise $S_{mpnb}(\sigma_1)$ would be incorrect or $S_{mpnb}(\sigma_2)$ would not be maximally permissive. Hence, S_{mpnb} is a state-based supervisor. □

The consequence of Theorem 3 is that in order to solve SSCP, it suffices to search over state-based supervisors. The state-based supervisor that satisfies the requirements of SSCP among all state-based supervisors will be equal to S_{mpnb}, the solution of SSCP.

As an illustration of state-based supervisors, consider our running example, and supervisors S_1 and S_2 defined in Section 2.1.3 and illustrated in Fig. 3. Both S_1 and S_2 are state-based and can be equivalently defined as follows:

$$S_1(x) = \{c_1, u\} \text{ for all } x \in \{x_0, x_1, x_2, x_3\} \quad S_2(x) = \begin{cases} \{c_1, u\} & \text{if } x = x_0 \\ \{c_2, u\} & \text{if } x \in \{x_1, x_2, x_3\} \end{cases}$$

Also consider supervisor S_3 defined in Section 2.1.6. S_3 is also state-based and can be equivalently defined as follows:

$$S_3(x) = \begin{cases} \{u\} & \text{if } x = x_0 \\ \{c_2, u\} & \text{if } x \in \{x_1, x_2, x_3\} \end{cases}$$

3.2 Reactive synthesis with plants

Most classical reactive synthesis frameworks (Manna and Wolper 1984; Pnueli and Rosner 1989a) do not have a notion of plant. In Pnueli and Rosner (1989a), the realizability problem is defined as the problem of synthesizing, given a temporal logic specification ϕ, an input-output strategy that implements ϕ. This is also how the reactive synthesis problem (RSP) is defined in Section 2.2. An exception to the above is the work of Madhusudan (2001), where the *control problem for non-reactive environments*[2] is defined as the problem of synthesizing a controller for a given plant modeled as a finite-state Kripke structure, so that the closed-loop system satisfies a specification in CTL or CTL*.

In view of building a bridge between supervisory control and reactive synthesis, in this section we recall Madhusudan's reactive synthesis problem with an explicit notion of plant, giving it the name *reactive synthesis control problem* (RSCP). In Section 3.5 we discuss links between RSCP and RSP.

3.2.1 Plants as kripke structures

As done in Madhusudan (2001), a plant can be captured as a *transition system*, specifically a form of *Kripke structure*:

$$P = (W, w_0, R, AP, L)$$

where

- AP is a set of atomic propositions.
- W is a set of states, $w_0 \in W$ being the initial state. W is (implicitly) partitioned into two disjoint subsets

  $$W = W_s \cup W_e$$

  W_s models the system states (where the system must choose a move). W_e models the environment states (where the environment must choose a move).
- $R \subseteq W \times W$ is the transition relation.
- $L : W \to 2^{AP}$ is a labeling function mapping every state w to a set of propositions true in this state. L must be total.

We assume that R is total, that is, for any $w \in W$, there exists $w' \in W$ such that $(w, w') \in R$. We define $succ_P(w) = \{w' \mid (w, w') \in R\}$. Because R is total, $succ_P(w) \neq \emptyset$ for all $w \in W$. When P is clear from context we write $succ$ instead of $succ_P$.

As an example, consider the Kripke structure P_1 shown in Fig. 11. States drawn as circles are system states. The square state is an environment state (it is also the initial state). The arrows between states represent the transition relation. Notice that there is at least one outgoing transition from every state, which ensures that the transition relation is total. P_1 has a single atomic proposition p holding only at state 3.

A Kripke structure plant is called finite when its set of states is finite.

[2]In Madhusudan (2001) this is also called the control problem for the *universal environment*. In his thesis, Madhusudan also defines a *control problem for reactive environments*, where the goal is to find a controller that works against *all possible strategies of the environment*, instead of a controller that works against the single, "maximally nondeterministic" strategy of the environment which is to offer all possible inputs (the latter is the universal environment). In the case of LTL specifications, a winning strategy for the maximally-nondeterministic environment is also winning for any other environment. As pointed out in Madhusudan (2001), this is not the case for CTL or CTL* specifications. For example, a specification of the form $\mathsf{E}\phi$ may be satisfied in a maximally-nondeterministic environment which allows a certain path satisfying ϕ, whereas in a more restrictive environment which does not allow such a path, the formula may not hold. Madhusudan (2001) shows that the control problem for reactive environments is harder (from a complexity point of view) than the control problem for the universal environment. For our purposes, the latter problem suffices to capture SSCP.

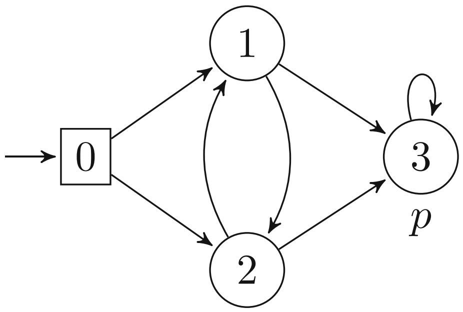


Fig. 11 Kripke structure P_1

3.2.2 *Using CTL* for Kripke Structures*

We have introduced the temporal logic CTL* for the specification of a system to be synthesized in Section 2.2.2. This logic is equally useful for specifying control objectives in plants. In this context, we evaluate the CTL* formula on the tree that is *induced* by the Kripke structure.

Let $P = (W, w_0, R, AP, L)$ be a Kripke structure. We say that P induces a computation tree $\langle T, \tau \rangle$ if the following conditions hold:

- $T \subseteq W^*$
- $\{t \in T : |t| = 1\} = \{w \in W : (w_0, w) \in R\}$
- $\tau(\epsilon) = L(w_0)$
- For all $t = t_0 t_1 \dots t_n \in T$, the set of $t's$ children is precisely $\{t_0 t_1 \dots t_n t_{n+1} \mid t_{n+1} \in W, (t_n, t_{n+1}) \in R\}$
- For all $t = t_0 t_1 \dots t_n \in T$, we have $\tau(t) = L(t_n)$.

In a nutshell, the computation tree that is induced by a Kripke structure represents all possible paths in the Kripke structure at the same time. A path of P is an infinite sequence $\pi = w_0 w_1 \cdots$, such that $w_i \in W$ and $(w_i, w_{i+1}) \in R$, for all $i \geq 0$.

Given some CTL* state formula ϕ, we say that some state $w \in W$ satisfies ϕ if the computation tree for the Kripke structure $P_w = (W, w, R, AP, L)$ that only differs from P by its initial state, satisfies ϕ. We say that a plant satisfies a CTL* state formula ϕ, written formally as $P \models \phi$, if the tree induced by P satisfies ϕ.

3.2.3 *Strategies*

A plant P may not generally satisfy a CTL* specification ϕ. A strategy aims to restrict P so that it satisfies ϕ. Let $P = (W, w_0, R, AP, L)$ with $W = W_s \cup W_e$. A *strategy for P* is a (total) function

$$f : W^* \times W_s \to 2^W$$

such that for all $u \in W^*$, $w \in W_s$, $f(u, w)$ is a non-empty subset of $succ(w)$. The intuition is that f observes the history of all states visited previously, $u \in W^*$, as well as the current system state $w \in W_s$, and chooses to allow moves to only a subset (but a non-empty subset) of the successors of w.

A strategy f is *state-based* if for all $u_1, u_2 \in W^*$, and for all $w \in W_s$, we have $f(u_1, w) = f(u_2, w)$. This means that the strategy only depends on the current state w and not on the previous history u.

A strategy f defines a new (infinite-state) Kripke structure P^f:

$$P^f = (W^f, w_0^f, R^f, AP, L^f)$$

where

- $W^f = W^* \times W$
- $w_0^f = (\epsilon, w_0)$
- $R^f = \{\big((u, w), (u \cdot w, w')\big) \mid \big(w \in W_e \wedge (w, w') \in R\big) \vee \big(w \in W_s \wedge w' \in f(u, w)\big)\}$
- $L^f(u, w) = L(w)$ for all $u \in W^*$, $w \in W$.

Note that R^f is guaranteed to be total. This is because R is assumed to be total, and f is required to be such that $f(u, w) \neq \emptyset$.

Some strategies for P_1 are shown in Fig. 12. The strategies f_1, f_2, f_3 are state-based, where f_2 disables the transition $2 \to 1$, f_3 disables the transition $1 \to 2$, and f_1 disables both of these transitions.

3.2.4 *Reactive Synthesis Control Problem (RSCP)*

Given Kripke structure plant P and CTL* formula ϕ, we say that a strategy f *enforces* ϕ *on* P if it is the case that $P^f \models \phi$. The *reactive synthesis control problem* (RSCP) is the following:

Definition 6 (RSCP) Given finite Kripke structure plant P and CTL* formula ϕ, find (if it exists, or state that there does not exist) a strategy which enforces ϕ on P.

RSCP-CTL denotes RSCP where ϕ is required to be a CTL formula; RSCP-LTL denotes RSCP where ϕ is required to be an LTL formula. (Similarly for RSP.)

3.2.5 *Maximal permissiveness in RSCP*

The reader may have observed that in the definition of RSCP above, we did not require that the strategy f be maximally-permissive in any way. The reason is that unique maximally-permissive strategies do not always exist. An example is given below. Let us first introduce some terminology.

Let f_1, f_2 be two strategies for a plant P. f_1 is said to be *no more permissive than* f_2 iff for all $u \in W^*$, $w \in W_s$ such that uw is a sequence of states that can be a prefix of a run in P^{f_2}, $f_1(u, w) \subseteq f_2(u, w)$. f_2 is said to be *strictly more permissive than* f_1 if f_1 is no more permissive than f_2 and $f_1(u, w) \neq f_2(u, w)$ for some $u \in W^*$, $w \in W_s$ such that uw is a sequence of states that can be a prefix of a run in P^{f_2}.

f_1 is said to be *maximally permissive* with respect to specification ϕ if f_1 enforces ϕ and there is no strategy f_2 which enforces ϕ and is strictly more permissive than f_1.

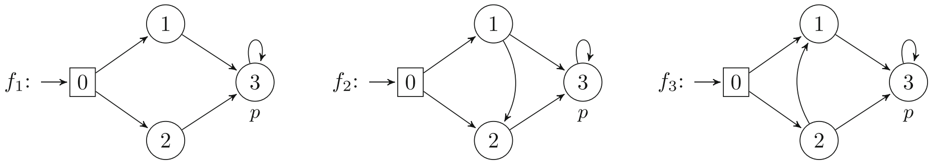


Fig. 12 Some strategies enforcing $\mathsf{AF}p$ on the Kripke structure P_1 of Fig. 11

Let us return to the example of Figs. 11 and 12. Suppose we wish to find a strategy that makes the plant P_1 meet the specification $\mathsf{AF}p$. The latter states that all executions must eventually reach a state satisfying p. Since state 3 is the only state satisfying p in P_1, we want all executions to reach state 3. On its own, P_1 does not satisfy $\mathsf{AF}p$, because it contains two executions "oscillating" between states 1 and 2.

All three strategies f_1, f_2, f_3 of Fig. 12 enforce $\mathsf{AF}p$ on P_1. Strategies f_2 and f_3 are strictly more permissive than f_1, and are the two (incomparable) most permissive state-based strategies for $\mathsf{AF}p$. However, there are infinitely many other, more permissive strategies which also enforce $\mathsf{AF}p$, not shown in Fig. 12. In particular, any (non-state-based) strategy which allows a finite number of transitions between states 1 and 2 before forbidding them, enforces $\mathsf{AF}p$.

There is a set of increasingly permissive such strategies, but the limit of this set is the strategy that forbids nothing, and this strategy no longer enforces $\mathsf{AF}p$. This example shows that a unique maximally-permissive strategy does not generally exist for the RSCP problem.

In Section 3.3.2, we will be concerned with CTL* specifications of the form $\mathsf{AGEF}q$, where q is a CTL* or CTL state formula without any temporal operator. For these, maximally-permissive strategies do exist, and they are at the same time state-based. Let us conclude the discussion of maximal permissiveness by proving this fact.

Lemma 2 *Let $P = (W, w_0, R, AP, L)$ be a Kripke structure, and q be a CTL state formula without temporal operators. If there exists a strategy enforcing $\mathsf{AGEF}q$ on P, then there exists a unique, maximally-permissive strategy enforcing $\mathsf{AGEF}q$ on P; moreover, this strategy is state-based.*

Proof Let A be the set of states of P from which there is no state reachable in P that satisfies q. Then any strategy f that allows to eventually visit a state of A in P^f cannot induce a computation tree that satisfies $\mathsf{AGEF}q$, as there exists at least one node in the tree from which $\mathsf{EF}q$ is false (i.e., one node in A).

On the other hand, if a strategy only leads to visiting states other than A, then every node in the computation tree for P^f satisfies $\mathsf{EF}q$ (by the definition of A). Therefore, in order for P^f to satisfy $\mathsf{AGEF}q$, it suffices for the strategy f to avoid visiting a state in A.

We prove that there exists a maximally-permissive, state-based strategy by showing that we can compute a set of *bad states* B that over-approximates A, and precisely the strategies that stay out of B lead to never having a path in P^f that eventually visits A.

We build B gradually, starting with A. We add some state $w \in W_s$ to B whenever there exists no successor of w that is not in B yet. Furthermore, we add some state $w \in W_e$ to B whenever there is some successor of w that is in B. As we only add states to B in this process, and never remove states, B converges to some well-defined set.

It can be shown by induction over the step of the computation in which a state is added to B that from all of them, $\mathsf{EF}q$ cannot be enforced. For step 0, in which we initialize B with A, the claim is trivial. For step $i+1$, we can assume that the claim is true for $i \in \mathbb{N}$. If there is a state $w \in W_s$ from which no matter what the strategy does, we land in a state in B, then obviously, once a run entered B, we cannot avoid to visit a state in A in the future (by induction), so adding w to B is justified. Likewise, if there is a state $w \in W_e$ with a successor in B, then we cannot avoid that after one step, we land in a state in B, so adding w to B is again justified.

Now assume that we have a strategy of the following form: we set $f(\overline{w}, w) = \{w' \in W \mid (w, w') \in R, w' \notin B\}$ if $w \notin B$, and $f(\overline{w}, w) = \{w' \in W \mid (w, w') \in R\}$ otherwise. By definition, there is no strategy f' that has $f(\overline{w}, w) \subset f'(\overline{w}, w)$ for some $\overline{w}$ and w such that $\overline{w}w$ is a path in the run tree of $P^{f'}$ and at the same time prevents visiting B from a state that is not in B, as it allows any transition to states not in B. As we have proven already that any strategy that allows visiting B from w_0 cannot enforce $\mathsf{AGEF}q$, there is no strategy that allows a move that is not allowed by f and enforces $\mathsf{AGEF}q$. Thus, f is the unique maximally-permissive strategy enforcing $\mathsf{AGEF}q$ if f is actually winning. Note that f is state-based, too.

It remains to prove that f is winning (i.e., f is not "too permissive"). We show this by induction over the length of a path in the run tree for P^f. As long as along no such path, we ever reach a state from B, all nodes in the computation tree satisfy $\mathsf{EF}q$. We start with the root of the computation tree of P^f and know already that if $w_0 \in B$, then there is no strategy to enforce $\mathsf{AGEF}q$. So we can assume that $w_0 \notin B$. For the inductive step, take some node $\overline{w}w$ that is not in B (by the inductive hypothesis). If $w \in W_e$, then by the definition of B, we have that all successors of w in P^f are also not in B. So we have that for all computation tree nodes $\overline{w}ww'$ of P^f with $w, w' \in W$ that $\overline{w}ww' \notin B$. If $w \in W_s$, then as f restricts P_f to the successor states that are not in B, the claim holds in this case as well. □

In view of reducing supervisory control problems to reactive synthesis problems, we would like to reduce SSCP to RSCP. However, this reduction cannot be done directly, because SSCP asks for a maximally-permissive supervisor, whereas RSCP only asks for a strategy (since a maximally-permissive strategy may not generally exist). To avoid this problem, we exploit the result of Lemma 2 and introduce a new problem, called $\text{RSCP}_{\max}^{\mathsf{AGEF}q}$, which is a variant of RSCP, and more precisely a variant of RSCP-CTL. In $\text{RSCP}_{\max}^{\mathsf{AGEF}q}$, the specification is a CTL formula of the form $\mathsf{AGEF}q$ where q is a CTL formula without temporal operators. Lemma 2 shows that for this class of specifications, existence of a strategy implies existence of a unique maximally-permissive strategy. $\text{RSCP}_{\max}^{\mathsf{AGEF}q}$ asks precisely for this strategy, if it exists.

Definition 7 ($\text{RSCP}_{\max}^{\mathsf{AGEF}q}$) Given finite Kripke structure plant P and CTL formula ϕ of the form $\mathsf{AGEF}q$ where q is a CTL formula without temporal operators, find (if it exists, or state that there does not exist) the unique maximally-permissive state-based strategy that enforces ϕ on P.

We can generalize $\text{RSCP}_{\max}^{\mathsf{AGEF}q}$ to a more general (and ambitious) reactive synthesis control problem with maximal-permissiveness. Although we currently do not know how to solve this problem, it is useful to define it as a way of motivating future work.

Definition 8 ($\text{RSCP}_{\max}$) Given finite Kripke structure plant P and CTL* formula ϕ, find, if there exists, a strategy which enforces ϕ on P. If so, find whether a maximally-permissive such strategy exists, and whether it is unique, and if so, compute it.

Other researchers have investigated problems related to $\text{RSCP}_{\max}$, in the context of related control frameworks and general logics; see, e.g., Riedweg and Pinchinat (2004), Pinchinat and Riedweg (2005), van Hulst et al. (2014). These works show that there are problem instances where maximally permissive solutions can be computed, when they exist.

3.3 From supervisory control to reactive synthesis with plants

In this section we show how to reduce SSCP to $\mathrm{RSCP}^{\mathsf{AGEF}q}_{\max}$.

3.3.1 From DES plants to Kripke structure plants

Given a plant G in the form of a DES, we first construct a plant P_G in the form of a Kripke structure. Consider plant G_1 of Fig. 3. The Kripke structure P_{G_1} is shown in Fig. 13. States drawn as circles are system states. States drawn as rectangles are environment states.

A system state of P_G is a state x of G. An environment state of P_G is either of the form (x, c), where $c \in E_c$, or $(x, \perp)$. All successors of system states are environment states, and vice versa. From a system state x, P_G has at most $|E_c|+1$ possible successors, one successor of the form (x, c) for each controllable event c which is enabled at state x in G, plus an extra successor $(x, \perp)$. Intuitively, choosing a subset of the successors of x amounts to allowing a subset of the controllable events enabled at x. If only $(x, \perp)$ is chosen, then all controllable events are disabled and only uncontrollable events (if any) are allowed to occur at x.

From environment state (x, c), P_G has an outgoing transition to a system state x' if either G has an uncontrollable transition from x to x', or G has a transition labeled c from x to x'. That is, the only transitions enabled from (x, c) are uncontrollable transitions or the controllable transition labeled c (there can only be one controllable transition labeled c, because G is deterministic). Note that if x has no controllable transition labeled c, then

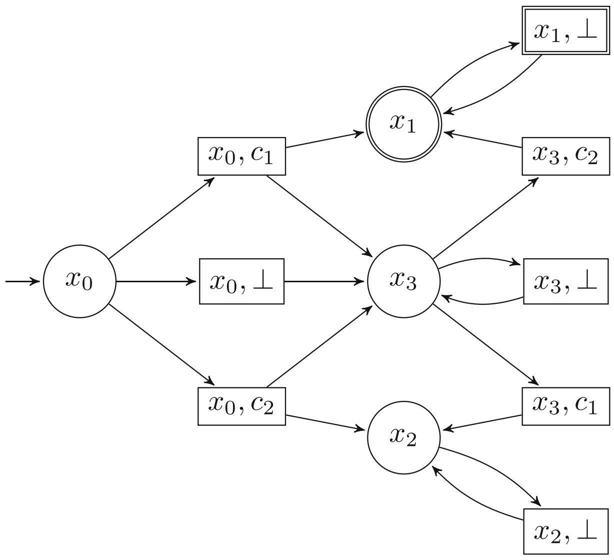


Fig. 13 Kripke structure P_{G_1} for DES G_1 of Fig. 3

(x, c) is not a successor of x by construction. Therefore, an outgoing transition is guaranteed to exist from every reachable environment state of the form (x, c) with $c \in E_c$.

Finally, from environment state $(x, \perp)$, P_G has an outgoing transition to a system state x' if G has an uncontrollable transition from x to x'. That is, only uncontrollable transitions are allowed from $(x, \perp)$. If x has no outgoing uncontrollable transitions then a transition back to x is added to $(x, \perp)$. In the example of Fig. 13, this is the case with states $(x_1, \perp)$, $(x_2, \perp)$, and $(x_3, \perp)$. These "back-transitions" achieve two goals. First, they prevent deadlocks in P_G. Second, they will allow us to prove that non-blocking strategies can always be extended to allow successors of the form $(x, \perp)$ (Lemma 4), a property which facilitates the arguments for maximal permissiveness.

Note that apart from back-transitions, whenever there is a transition from some environment state $(x, \perp)$ to some state x', then every environment state (x, c) for some c also has a transition to x'. This is due to the requirement that the uncontrollable transitions are always enabled. Only the back-transitions are not duplicated; due to the fact that (x, c) is only reachable if c is an enabled action in x, deadlock-freedom is already ensured without the back-transitions.

So far we have defined the states and transitions of P_G. We also need to define its set of atomic propositions and labeling function. P_G will have a single atomic proposition, acc. The states of P_G labeled with acc will be system states x which are marked states in G, and environment states (x, c) or $(x, \perp)$ where x is marked in G. In our example of Figure 13, this is the case with states x_1 and $(x_1, \perp)$, drawn with double lines to represent the fact that they are labeled with acc.

3.3.2 *Stating SSCP in temporal logic*

We now express the requirements of SSCP as a temporal logic formula. We will use the CTL formula

$$\phi_{nb} \;:=\; \mathsf{AG}\,\mathsf{EF}\,acc.$$

ϕ_{nb} states that it is always possible to reach a marked state, from any reachable state. This formula characterizes non-blockingness.

Returning to our example of Fig. 13, we observe that P_{G_1} does not satisfy ϕ_{nb} on its own: this is because from state x_2 there is no path reaching a state where acc holds. The same is true for state $(x_2, \perp)$. Therefore, in order to enforce ϕ_{nb}, a strategy must make these states unreachable.[3] Three such (state-based) strategies are shown in Fig. 14.

The interpretation of the two right-most strategies of Fig. 14 is quite clear. f_5 disables c_2 at state x_0 and c_1 at x_3. f_6 disables both c_1 and c_2 at x_0, and c_1 at x_3. Neither f_5 nor f_6 are maximally-permissive strategies for ϕ_{nb}. Strategy f_4, on the other hand, is maximally-permissive.

The interpretation of f_4 is perhaps puzzling. Interpreted as a supervisor, it appears to allow c_1 at x_0 (because it keeps the transition from x_0 to (x_0, c_1)), and at the same time to

[3]Note that from state $(x_3, \perp)$ there *is* a path to acc. This may seem counter-intuitive, as $(x_3, \perp)$ may be interpreted as the state where the plant is at x_3 and the supervisor has disabled all controllable events. Since no uncontrollable event exists at x_3, this must be a blocking situation. As it turns out, this is not a problem, because we insist on maximally-permissive supervisors and strategies. Such a strategy either allows also controllable events from x_3, in which case blockingness is avoided by reaching the corresponding controllable successors, or disables all controllable successors of x_3, in which case x_3 is already blocking. Therefore, it is safe to allow a back-transition from $(x_3, \perp)$ to x_3.

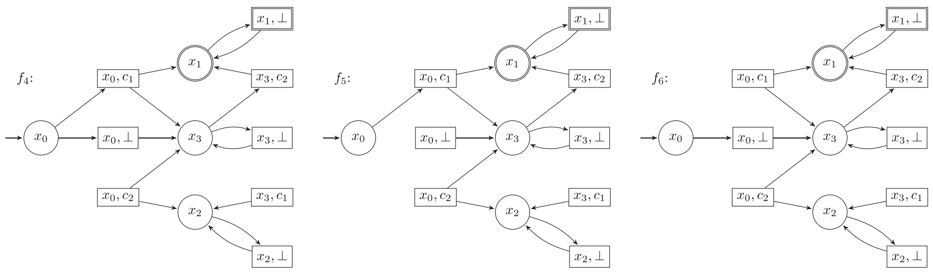


Fig. 14 Strategies enforcing ϕ_{nb} on P_{G_1} of Fig. 13

disable all controllable events at x_0 (because it keeps the transition from x_0 to $(x_0, \perp)$). We will not worry about this paradox, which we take to be only a matter of interpretation. The way P_G is defined, every successor of $(x, \perp)$ is also a successor of (x, c) for any $c \in E_c$. As a result, $(x, \perp)$ may be redundant, but it does not harm. On the contrary, it will allow us to prove existence of unique maximally-permissive strategies.

This intuition is formalized in Lemma 4 below.

3.3.3 The formal reduction

Let $G = (X, x_0, X_m, E, \delta)$ be a DES plant with $E = E_c \cup E_{uc}$. It is convenient to define the functions $\mathsf{En} : X \to 2^E$ with $\mathsf{En}(x) = \{e \mid \delta(x, e) \text{ is defined}\}$, $\mathsf{En}_c : X \to 2^{E_c}$ with $\mathsf{En}_c(x) = \mathsf{En}(x) \cap E_c$, and $\mathsf{En}_u : X \to 2^{E_{uc}}$ with $\mathsf{En}_u(x) = \mathsf{En}(x) \cap E_{uc}$. The functions $\mathsf{En}, \mathsf{En}_c, \mathsf{En}_u$ return, respectively, the set of all events, controllable events, and uncontrollable events, enabled at state x. For instance, a state $x \in X$ is a deadlock in G iff $\mathsf{En}(x) = \emptyset$.

The Kripke structure plant P_G is defined to be

$$P_G = (W, w_0, R, AP, L)$$

such that

- $W = W_s \cup W_e$, with $W_s = X$ and $W_e = X \times (E_c \cup \{\perp\})$.
- $w_0 = x_0$. Therefore, w_0 is a system state.
- $R = R_s \cup R_e$, with

$$\begin{aligned}
R_s \;=\; & \{(x, (x, c)) \mid x \in X, c \in \mathsf{En}_c(x)\} \;\cup\; \{(x, (x, \perp)) \mid x \in X\} \\
R_e \;=\; & \{\big((x, c), x'\big) \mid x, x' \in X, \exists e \in E_u \cup \{c\} : \delta(x, e) = x'\} \\
& \cup \{((x, c), x) \mid x \in X, c \in E_c, c \notin \mathsf{En}_c(x)\} \\
& \cup \{\big((x, \perp), x'\big) \mid x, x' \in X, \exists u \in E_u : \delta(x, u) = x'\} \\
& \cup \{((x, \perp), x) \mid x \in X, \mathsf{En}_u(x) = \emptyset\}
\end{aligned}$$

- $AP = \{acc\}$.
- $L(s) = \begin{cases} \{acc\} & \text{if } s = x \text{ or } s = (x, \perp) \text{ for some } x \in X_m \\ \{\} & \text{otherwise.} \end{cases}$

The following lemma guarantees that P_G does not have deadlocks, therefore, it is a valid Kripke structure plant as required in Section 3.2.1.

Lemma 3 *The transition relation R of P_G defined above is total.*

Proof Let $w \in W$ be a state of P_G. We need to find $w' \in W$ such that $(w, w') \in R$. We distinguish cases.

- Suppose $w \in W_s = X$. In this case, we have that $(w, (w, \bot)) \in R$ by definition of R_s.
- Suppose $w \in W_e = X \times (E_c \cup \{\bot\})$.
  - Suppose w is of the form (x, c), for $x \in X, c \in E_c$.
    - If $\mathsf{En}_u(x) \neq \emptyset$ then there exists $x' \in X$ and $u \in E_{uc}$ such that $\delta(x, u) = x'$. Then $\big((x, c), x'\big) \in R$ by definition of R_e.
    - If $\mathsf{En}_u(x) = \emptyset$ then
      - if $\delta(x, c)$ is defined and equals x', then $\big((x, c), x'\big) \in R$ by definition of R_e;
      - otherwise, $c \notin \mathsf{En}_c(x)$, therefore $((x, c), x) \in R$ by definition of R_e.
  - Suppose w is of the form $(x, \bot)$, for $x \in X$.
    - If $\mathsf{En}_u(x) \neq \emptyset$ then there exists $x' \in X$ and $u \in E_u$ such that $\delta(x, u) = x'$. Then $\big((x, \bot), x'\big) \in R$ by definition of R_e.
    - If $\mathsf{En}_u(x) = \emptyset$ then $((x, \bot), x) \in R$ by definition of R_e.

□

In the sequel we simplify notation for state-based strategies as follows. Whenever we are concerned with a state-based strategy f for some set of states W, we simply write $f(w)$ for some $w \in W_s$ to mean the value of $f(\overline{w}, w)$ for any $\overline{w} \in W^*$. Since for state-based strategies the value of $\overline{w}$ does not make a difference, $f(w)$ is uniquely defined.

Definition 9 Let $G = (X, x_0, X_m, E, \delta)$ be a DES plant and $P_G = (W, w_0, R, AP, L)$ be a Kripke structure built from G by the construction above. Let f be a state-based strategy for P_G. The *$\bot$-closure of f* is defined to be the state-based strategy f' that results from setting $f'(x) = f(x) \cup \{(x, \bot)\}$ for all $x \in W_s$.

Lemma 4 *Let $G = (X, x_0, X_m, E, \delta)$ be a DES plant and $P_G = (W, w_0, R, AP, L)$ be a Kripke structure built from G by the construction above. Let f be a strategy enforcing ϕ_{nb} on P_G. Let f' be the $\bot$-closure of f. Then f' also enforces ϕ_{nb} on P_G.*

Proof The specification ϕ_{nb} is $\mathsf{AGEF}acc$. This formula can only hold in a node of the computation tree if from every node in the computation tree, some other node marked acc can be reached. For every state-based strategy f and $x \in W_s$, we have that if $f'(x) \neq f(x)$, then there has to exist some $(x, c) \in f(x)$. This is because $f(x)$ must be non-empty and if all it contains is $(x, \bot)$ then we would have $f'(x) = f(x)$. As $(x, \bot)$'s successors are always a subset of (x, c)'s successors, and adding paths to nodes that already satisfy $\mathsf{AGEF}acc$ does not change the fact that the computation tree satisfies $\mathsf{AGEF}acc$, such a modification of the strategy does not alter the fact that it induces a computation tree that satisfies ϕ_{nb}. □

The next theorem shows a one-to-one correspondence between maximally permissive strategies in a DES plant G and maximally permissive strategies in the Kripke structure P_G

built from G according to the definition at the beginning of this section. Fig. 15 exemplifies the idea in the context of plant G_1 presented earlier in Fig. 3. The figure shows the two respective strategies, one for G_1 and one for its corresponding Krikpe structure. Every transition disabled in the strategy for G_1 has a corresponding transition disabled in the Kripke structure (and vice versa). Recall from above that the Kripke structure has as state set $W = W_s \cup W_e$ for $W_s = X$, where X is the set of the states in the DES plant. The strategies have the property that states $x \in X$ are reachable in the DES plant if and only if they are reachable in the Kripke structure. Also, the strategy in the Kripke structure never disables transitions to states of the form $(x_0, \perp)$, as the outgoing edges in $(x_0, \perp)$ represent uncontrollable transitions in the plant. Since by the construction presented at the beginning of this section the outgoing edges of a state of the form $(x, \perp)$ are always a subset of the edges from states of the form (x, c) for the same value of x, a maximally permissive strategy in the Kripke structure that does not deadlock in a state x thus does not need to deactivate the transition to $(x, \perp)$.

Theorem 4 *Let $G = (X, x_0, X_m, E, \delta)$ be a DES plant and $P_G = (W, w_0, R, AP, L)$ be a Kripke structure built from G by the construction above.*

1. *Given a non-blocking maximally-permissive state-based supervisor $S : X \to 2^E$ for G, we can compute a maximally-permissive state-based strategy f_S enforcing* AGEF*acc on P_G as follows:*

$$\text{For all } w \in W_s,\ f_S(w) = \{(w, c) \mid c \in S(w) \cap E_c\} \cup \{(w, \perp)\}.$$

2. *Given a maximally-permissive state-based strategy f enforcing* AGEF*acc on P_G, we can compute a non-blocking state-based maximally-permissive supervisor S_f for G as follows:*

$$\text{For all } x \in X,\ S_f(x) = E_{uc} \cup \{e \in E_c \mid (x, e) \in f(x)\}.$$

Proof We prove this result in three steps:

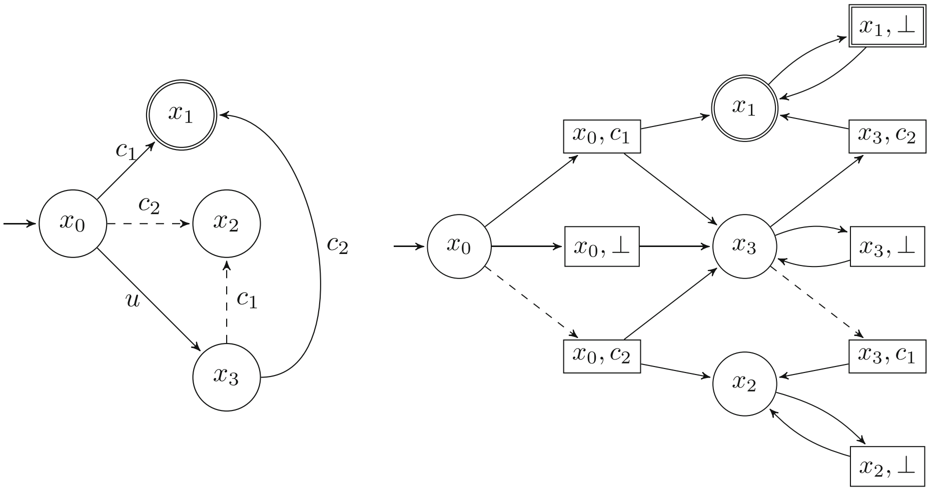


Fig. 15 DES plant G_1 (the same as in Figure 3), its corresponding Kripke structure, and a strategy for G_1 that is mapped to its Kripke structure. Dashed lines represent transitions that are disabled by the strategies

(a) For every state-based supervisor S which is non-blocking for G, the state-based strategy f_S defined above is $\perp$-closed and enforces $\mathsf{AGEF}acc$ on P_G.
(b) Starting from a $\perp$-closed state-based strategy f which enforces $\mathsf{AGEF}acc$ on P_G, the supervisor S_f defined above is a state-based supervisor which is non-blocking for G.
(c) Translating from a $\perp$-closed strategy f to S and back will yield the same strategy as we started with, i.e., $f_{S_f} = f$. Furthermore, if f is a strategy that is strictly more permissive than some strategy f', then S_f is strictly more permissive than $S_{f'}$. Likewise, if S is strictly more permissive than S', then f_S is strictly more permissive than $f_{S'}$.

Taking these facts together, we obtain that the translations above must map the (unique) maximally-permissive non-blocking supervisor S for G and the maximally-permissive strategy enforcing ϕ_{nb} on P_G onto each other. Otherwise, mapping one of the maximal solutions would yield a "more maximal solution" that is still a valid strategy/supervisor, which contradicts maximal permissiveness.

More precisely, facts (a) and (c) imply part 1 of the theorem. Indeed, let S^* be the (unique) state-based maximally-permissive non-blocking supervisory for G and let $f^* = f_{S^*}$. By definition f^* is state-based. Also, by (a), we know that f^* enforces ϕ_{nb} on P_G. To prove part 1 of the theorem, it remains to show that f^* is maximally-permissive. Suppose not. Then there exists strategy f which enforces ϕ_{nb} on P_G and is strictly more permissive than f^*. By (a), f^* is $\perp$-closed, therefore f must also be $\perp$-closed. Let $S = S_f$. By (c), S must be strictly more permissive than S^*, which contradicts maximal permissiveness of the latter.

Similarly, facts (b) and (c) imply part 2 of the theorem. Indeed, let f^* be the unique (by Lemma 2) state-based maximally-permissive strategy enforcing ϕ_{nb} on P_G. Lemma 4 and maximal permissiveness of f^* ensure that f^* is $\perp$-closed. Let $S^* = S_{f^*}$. By definition S^* is state-based. Also, by (b), we know that S^* is non-blocking for G. It remains to show that S^* is maximally-permissive. Suppose not. Then there exists non-blocking supervisor S for G which is strictly more permissive than S^*. Let $f = f_S$. By (c), f must be strictly more permissive than f^*, which contradicts maximal permissiveness of the latter.

It remains to perform the three steps outlined above.

Step (a): We prove two sub-steps. First of all, we show that for every state s that is reachable in S/G, from any state $(\overline{w}, s)$ in P_G^f, we can reach an acc-labeled state. This implies that $\mathsf{EF}acc$ holds at state $(\overline{w}, s)$. Then, we show that the only reachable states $(\overline{w}, s)$ in P_G^f are those that have a successor state $(\overline{w}, s')$ for which s' is reachable in S/G, or s is reachable in S/G. This shows that for all reachable states in P_G^f, $\mathsf{EF}acc$ holds, and thus, the computation tree of P_G^f satisfies $\mathsf{AGEF}acc$. The fact that f is $\perp$-closed is trivial as it is $\perp$-closed by definition.

For the first sub-step, consider a state $s \in S$ that is reachable in S/G. As S is a non-blocking supervisor for G, we have that there exists some sequence $s_1e_1s_2e_2\ldots e_{n-1}s_n$ in S/G with $s_1 = s$, $L(s_n) = acc$, and for every $i \in \{1, \ldots, n-1\}$, we have that $s_{i+1} = \delta(s_i, e_i)$.

The construction of f and P_G^f makes sure that $\pi = s_1(s_1, c_1)s_2(s_2, c_2)\ldots(s_{n-1}, c_{n-1})s_n$ is a valid path in P_G for $c_i = e_i$ whenever $e_i \in E_c$, and $c_i = \perp$ otherwise, and for every node $(\overline{w}, s)$ in P_G^f, there is a path from $(\overline{w}, s)$ whose projection is π.

The second sub-step is proven by induction over a run in P_G^f. We start with state $(\epsilon, w_0) = (\epsilon, x_0)$, for which x_0 in S/G is reachable by definition. For the induction step, assume that the claim holds for all previous states $(\overline{w}, w)$ for $w \in W$. If we have that $w = x$ for some $x \in X$, then all possible successors in P_G^f can only be of the form $(\overline{w}', (x, c))$, so the claim holds here as well. If on the other hand we have that $w = (x, c)$ for some $x \in X$, then either we have $c = \bot$, or we have $c \in E_c$ with $c \in S(x)$, as we defined f to only allow these actions, and thus, these are the only successors that occur in P_G^f. If $c = \bot$, then the next element of the run in P_G^f can only be either $(\overline{w}', x)$, in which case the claim holds, or $(\overline{w}', x')$ for some x' with $x' = \delta(x, c')$ for some $c' \in E_u$. As by the definition of a supervisor for controlling a plant, the supervisor cannot deactivate uncontrollable actions, if x is reachable, then x' is also reachable, and thus the claim holds here, too. On the other hand if $c \neq \bot$, then the next element of the run in P_G^f can only be either $(\overline{w}', (x, c))$ or x' with $x' = \delta(x, c)$. In the first case, the claim holds again, and in the second, as c can only have been selected if $c \in S(x)$, we need to have that the transition from x to x' is possible in S/G, too, so again, the claim holds in this case.

Step (b): We prove two sub-steps. First of all, we show that for every state $(\overline{w}, x)$ for some $x \in X$ that is reachable in P_G^f, from state x in S_f/G, we can reach an accepting state. Then, we show that for every reachable states s in S_f/G, there is some reachable state $(\overline{w}, s)$ in P_G^f. This shows that from all states that are reachable in S_f/G, there is some path to an accepting state, which implies that S_f is a valid supervisor for G.

For sub-step 1, let $(\overline{w}, x)$ be some reachable state in P_G^f. As P_G^f satisfies $\mathsf{AGEF}acc$, there has to exist some path π from $(\overline{w}, x)$ to some $(\overline{w}', x')$ with $L((\overline{w}', x')) = acc$. Without loss of generality, let this path be loop-free (as we can always cut out loops in the path). We build from π a path from x to x' in S_f/G. Whenever we move along this path from some state $(\overline{w}'', x)$ to some state $(\overline{w}''', (x, c))$ or to $(\overline{w}''', (x, \bot))$, we do not add a step in the path for S_f/G. On the other hand, whenever we move from some state $(\overline{w}'', (x, c))$ or $(\overline{w}'', (x, \bot))$ to $(\overline{w}''', x')$, we add x' to the path to be constructed. By the definition of S_f, this is always a transition that is allowed by S_f. At the end of the path, as we have $L((\overline{w}', x')) = acc$ if and only if x' is marked, our reconstructed path in S_f/G ends in a marked state.

For sub-step 2, we reconstruct a path in P_G^f from a path $\pi = x_0 \dots x_n \in X^\omega$ in S_f/G. Let $\rho = \rho_0 \dots \rho_{n-1}$ be the actions to create the path π, i.e., we have $x_{i+1} \in \delta(x_i, \rho_i)$ for every $i \in \mathbb{N}$. We have $\rho_i \in f(x_i)$ by the definition of S for every $i \in \mathbb{N}$. This allows us to construct the path $\pi' = (\epsilon, x_0)(x_0, (x_0, c_0))(x_0(x_0, c_0), x_1)(x_0(x_0, c_0)x_1, (x_1, c_1)) \dots$ in P_G^f, where for every $i \in \mathbb{N}$, we have $c_i = \bot$ if ρ_i is an uncontrollable action, and $c_i = \rho_i$ otherwise. As π will end in $(\overline{w}, x_n)$ for some $\overline{w}$, the sub-claim follows.

Step (c): The claim for step 3 consists of three sub-steps. For the first sub-step, let f be a $\bot$-closed strategy, and S_f be the corresponding supervisor. We translate S_f back tlo a state-based supervisor for P_G and obtain:

$$\begin{aligned} f_{S_f}(w) &= \{(w, c) \mid c \in S_f(w) \cap E_c\} \cup \{(w, \bot)\} \\ &= \{(w, c) \mid c \in \{e \in E_c \mid (w, c) \in f(w)\}\} \cup \{(w, \bot)\} \\ &= f(w) \end{aligned}$$

For the second sub-step, let f and f' be state-based strategies such that for all $w \in W$, we have $f'(w) \subseteq f(w)$ and for one $w \in W$, we have $f'(w) \subset f(w)$. In this case, we have for all $w' \in W \setminus \{w\}$:

$$\begin{aligned} S_{f'}(w') &= E_{uc} \cup \{e \in E_c \mid (w', e) \in f'(w')\} \\ &\subseteq E_{uc} \cup \{e \in E_c \mid (w', e) \in f(w')\} \\ &= S_f(w') \end{aligned}$$

Furthermore, we have:

$$\begin{aligned} S_{f'}(w) &= E_{uc} \cup \{e \in E_c \mid (w, e) \in f'(w)\} \\ &\subset E_{uc} \cup \{e \in E_c \mid (w, e) \in f(w)\} \\ &= S_f(w) \end{aligned}$$

From line two to line three in this equation, we used the fact that the elements of $f'(w)$ are *all* of the form (w, c) for some $c \in E_c \cup \{\bot\}$ (except if $\mathsf{En}(w) = \emptyset$, in which case $f'(w) \subset f(w)$ cannot be fulfilled as we need to have $f'(w) = f(w) = (w, \bot)$ then) and by assumption, we have that $(w', \bot) \in f'(w')$, so there has to exist some $c \in \mathsf{En}_c(w)$ with $(w, c) \in f(w)$ but $(w', c) \notin f'(w)$ for all $w' \in W$.

For the third sub-step, we apply the same idea as in sub-step two. Let S and S' be state-based supervisors such that for all $x \in X$, we have $S'(x) \subseteq S(x)$ and for one $x \in X$, we have $S'(x) \subset S(x)$. Then for all $x \in X \setminus \{x\}$, we have:

$$\begin{aligned} f_{S'}(x') &= \{(x', c) \mid c \in S'(x') \cap E_c\} \cup \{(w, \bot)\} \\ &\subseteq \{(x', c) \mid c \in S(x') \cap E_c\} \cup \{(w, \bot)\} \\ &= f_S(x') \end{aligned}$$

Furthermore, we have:

$$\begin{aligned} f_{S'}(x) &= \{(x, c) \mid c \in S'(x) \cap E_c\} \cup \{(w, \bot)\} \\ &\subset \{(x, c) \mid c \in S(x) \cap E_c\} \cup \{(w, \bot)\} \\ &= f_S(x) \end{aligned}$$

This result completes step three over the overall proof.

□

Corollary 2 *SSCP can be reduced to RSCP$_{\max}^{\mathsf{AGEF}q}$ with a polynomial-time reduction.*

Proof Theorem 4 establishes a one-to-one correspondence between maximally-permissive state-based non-blocking supervisors for a DES plant G and a maximally-permissive state-based strategy for $\mathsf{AGEF}acc$ in a Kripke structure P_G that we construct from G. It follows that we can reduce the search for a supervisor in G as SSCP requests to searching for a strategy in P_G as RSCP$_{\max}^{\mathsf{AGEF}q}$ requests. Moreover, P_G can be constructed from G in polynomial time: the number of states of P_G is $O(n \cdot (m + 2))$ where n is the number of states of G and m is the number of controllable events. □

3.4 Algorithms

3.4.1 Algorithms for supervisory control problems

The formulation and solution of BSCP-NB were first presented in the seminal papers of Ramadge and Wonham (1987), Wonham and Ramadge (1987). The "standard" algorithm for solving BSCP-NB builds a non-blocking automaton that marks the language $\mathcal{L}_m(S_{mpnb}/G) = L_{am}^{mpnb}$ in the notation of Section 2.1.7 from G and from a non-blocking automaton that marks the language L_{am}. Let H be a deterministic automaton such that $\mathcal{L}(H) = \overline{L_{am}}$ and $\mathcal{L}_m(H) = L_{am}$. Let G have n states and H have m states. The standard algorithm for BSCP-NB builds automaton H^{mpnb} such that $\mathcal{L}(H^{mpnb}) = \overline{L_{am}^{mpnb}}$ and $\mathcal{L}_m(H^{mpnb}) = L_{am}^{mpnb}$ by first forming the product of G with H, and then iterating over the resulting structure to delete states that violate the safety property with an uncontrollable event and/or are blocking. Iterations are necessary in general since deletion of states that violate the safety property with an uncontrollable transition may create new blocking states, and vice-versa. Convergence is guaranteed in a finite number of steps since the number of states is finite. Hence, the computational complexity of the algorithm is $O(n^2m^2)$ in the worst case. This complexity does not include the construction of automaton H. There are special cases where the computational complexity can be reduced to $O(nm)$ in the worst case, such as when $L_{am} = \overline{L_{am}}$ or when all cycles in G contain a marked state.

Once H^{mpnb} has been obtained, it is effectively an encoding of a state-based (with respect to $G \times H$) supervisor that achieves the maximally permissive language L_{am}^{mpnb}; the transitions that are defined at each state of H^{mpnb}, which is a pair (x_G, x_H) with x_G a state of G and x_H a state of H, are the *enabled* controllable events and the feasible uncontrollable events.

Our transformation of BSCP-NB to SSCP in Section 3.1 is essentially an implementation of the first step of the standard algorithm (although we used an automaton A with a complete transition function therein).

The reader is referred to Cassandras and Lafortune (2008), Wonham (2015) for textbook expositions of the above material. To make this paper more self-contained, a simple algorithm for solving SSCP is given in Appendix C. This algorithm is a special case of the standard algorithm for solving BSCP-NB from Ramadge and Wonham (1987), Wonham and Ramadge (1987).

3.4.2 Algorithms for reactive synthesis problems

Algorithms for solving RSP-LTL have been provided in a number of works, e.g., in Pnueli and Rosner (1989a). Generally, these algorithms follow a similar flow, where the LTL formula ϕ is translated into some type of word automaton such as a Büchi automaton A_w, then A_w is translated into a tree automaton A_t, and finally A_t is translated into a game which is then solved algorithmically. Different methods differ by the type of automata and games that they use and how they represent them (e.g., enumeratively or symbolically). Also, in some cases some of the above steps may be missing as they are trivial. We refer the reader to Ehlers (2013) for a comprehensive overview.

Techniques for solving RSP-CTL and RSP are provided in a number of works, for instance, Kupferman and Vardi (1999), Madhusudan (2001). Madhusudan's thesis

(Madhusudan 2001) also provides a method for solving RSCP (and thus also RSCP-LTL and RSCP-CTL as special cases) by reducing it to the *module-checking problem* (Kupferman and Vardi 1996).

Because of the special form of the formula $\mathsf{AGEF}q$ and in view of the results in Section 3.3, $\mathrm{RSCP}^{\mathsf{AGEF}q}_{\max}$ can be solved using algorithms for supervisory control problems such as SSCP or BSCP-NB. As described in Appendix C, the general idea is to start from a set of *bad* states (initially those that violate $\mathsf{EF}q$) and iterate by labeling additional states as *bad*, if no strategy exists to avoid states that are already labeled *bad*. In the finite-state case, the algorithm ends when no more states can be added to the set of *bad* states, or when the initial state is added to that set. In the latter case, no winning strategy exists. The complexity of such an algorithm is polynomial in the number of states.

3.5 Reactive synthesis with plants vs. reactive synthesis without plants

In this section we discuss the links between the reactive synthesis problem with plants, RSCP, and the reactive synthesis problem without plants, RSP. Our discussion is informal, as it is beyond the scope of this paper to present detailed, formal reductions between these problems. Roughly speaking, RSP can be seen as a special case of RSCP, where the plant offers some possible input at every step. Some technical details need to be resolved, as RSP is formulated in terms of inputs and outputs whereas RSCP is formulated in terms of system and environment states. Bridging this gap should be relatively straightforward.

Conversely, RSP may appear at first sight more restrictive than it really is, as there is no notion of a plant that encodes the possible environment behavior. Yet, we can *encode* the plant behavior into the specification. Starting from a specification ϕ, we can modify it to some specification ϕ' such that in order for the controller to satisfy ϕ', it suffices that it satisfies ϕ only for those input streams that correspond to paths in a given plant. This roughly corresponds to ϕ' being of the form $\phi_{plant} \implies \phi$, where ϕ_{plant} models the behavior of the plant. In this way, a strategy for controlling a plant can be obtained by chopping away the irrelevant parts of a computation tree that satisfies ϕ'.

The above approach may at first appear uninteresting from a practical perspective, especially when a plant is already available in the form of an automaton (or a network of automata). Indeed, for most temporal logics the reactive synthesis problem is at least exponential in the length of the formula, so keeping the size of the formula small is essential. Moreover, an exponential blow-up typically occurs when translating the formula into some form of automaton during the synthesis procedure. Therefore, encoding a plant automaton to a temporal logic specification only to translate it back to an automaton during the synthesis process may seem computationally inefficient.

However, there are methods that successfully use the approach of encoding plants as temporal logic formulas, relying on the fact that sometimes the plant can be encoded in a certain fragment of temporal logic which does not suffer from exponential blow-up during translation to automata or synthesis. This is exploited, for instance, in Kress-Gazit et al. (2007) where synthesis is used for robotics applications such as motion planning. The plant encoded in this case is the region structure of the workspace, and it is encoded in the GR(1) fragment of LTL, which admits efficient synthesis algorithms (Piterman et al. 2006). In the case of Kress-Gazit et al. (2007), the overall approach is polynomial in the number of regions.

A related approach is taken in Pinchinat and Riedweg (2005), where an alternating automaton model is used that can capture the specification formula with only linear blow-up. Then the product of this automaton and the plant is computed resulting in an alternating automaton of the same type as the automaton for the formula. Because the automaton model is essentially as powerful and concise as the formula, this boils down to the same idea as encoding the plant into the specification.

4 Conclusion

This work is an introductory treatment of the connection between the two research fields of supervisory control and reactive synthesis. Although both fields target the general problem of controller synthesis for dynamic systems, they have developed largely independently over the last three decades, often emphasizing different, yet complementary, aspects of the general controller synthesis problem. The connection established in this paper was purposefully focused on the most basic problem of supervisory control, so that non-expert readers can get a general understanding of how the gap between these two fields can be bridged. At the same time, the formal reduction described in Section 3.3 is a novel contribution. It is our hope that this paper will motivate more in-depth studies on bridging this gap and more generally advancing the state-of-the-art in controller synthesis for discrete event systems by leveraging the techniques developed in both supervisory control and reactive synthesis.

As was mentioned in the introduction, connections between supervisory control and reactive synthesis have already been established in more advanced problem domains, such as partially-observed systems, languages of infinite strings, and distributed and decentralized control settings. Yet, a number of interesting topics for future work emerge from the more elementary connection established in this paper. One such topic is extending the results in this paper to multi-tasking supervisory control (see, e.g., de Queiroz et al. (2005)), where there are several sets of marked states that must each remain reachable.

Another topic is modeling and evaluation. We have only discussed illustrative examples in this paper. Although numerous interesting case studies in synthesis do exist in the literature in both supervisory control and reactive synthesis, these have limited comparative value as they were done in one of the two frameworks, but not in both. It would be worthwhile to develop case studies that would allow a detailed comparison of these two frameworks in terms of plant and specification modeling, computational complexity of synthesis, and implementation of derived supervisor/controller.

Acknowledgments We would like to thank Orna Kupferman, Madhusudan Parthasarathy, and John Thistle, for a number of interesting discussions, and Dror Fried for a careful reading of an earlier version of the paper. We also thank the anonymous reviewers for their pertinent comments.

A Proof of theorem 1

We use the formalism defined in Section 2.1 to show that the concept of *maximally permissive* solution, which is a central requirement in problem BSCP-NB of Section 2.1.8, is well defined. This will provide a proof of Theorem 1 that is self-contained.

For this purpose, we must first define the disjunction of two supervisors.

Let G be a DES plant and let S_1, S_2 be two supervisors for G. We define $S_1 \cup S_2$ to be a new supervisor, denoted by $S_{1\cup 2}$ for G, such that $S_{1\cup 2}(\sigma) = S_1(\sigma) \cup S_2(\sigma)$ for all $\sigma \in E^*$. We call $S_{1\cup 2}$ the *disjunction* of S_1 and S_2, since $S_{1\cup 2}$ allows all strings that S_1 and S_2 respectively allow.

We now wish to characterize the controlled behavior $\mathcal{L}(S_{1\cup 2}/G)$ under the disjunction of S_1 and S_2. Recall that when S_i is applied to G in isolation, the definition of S_i over $E^* \setminus \mathcal{L}(S_i/G)$ is irrelevant, since these strings will never occur in the controlled system. However, in the context of disjunction, this is no longer true since the controlled behavior will in general exceed $\mathcal{L}(S_i/G)$ due to the actions of the other supervisor(s). In order to allow for a simple characterization of $\mathcal{L}(S_{1\cup 2}/G)$, we make the following assumption:

$$S_i(\sigma) = E_{uc} \text{ for all } \sigma \in E^* \setminus \mathcal{L}(S_i/G). \tag{3}$$

We refer to supervisors that satisfy this assumption as "G-matched supervisors."

We can think of S_i as being "out of range" when the controlled language leaves $\mathcal{L}(S_i/G)$. Hence, when this occurs, S_i will not enable any events, other than uncontrollable ones, as it assumes that the controlled behavior is now in the range of another supervisor. Any supervisor S_i', when applied in isolation to G, can be replaced by a G-matched one, S_i, that results in the same controlled behavior $\mathcal{L}(S_i'/G) = \mathcal{L}(S_i/G)$.

It follows directly from the G-matched assumption and from the definition of disjunction that

$$\mathcal{L}(S_{1\cup 2}/G) = \mathcal{L}(S_1/G) \cup \mathcal{L}(S_2/G). \tag{4}$$

Since marking is a property of the plant, we similarly have that

$$\begin{aligned}
\mathcal{L}_m(S_{1\cup 2}/G) &= \mathcal{L}(S_{1\cup 2}/G) \cap \mathcal{L}_m(G) &(5)\\
&= [\mathcal{L}(S_1/G) \cup \mathcal{L}(S_2/G)] \cap \mathcal{L}_m(G) &(6)\\
&= [\mathcal{L}(S_1/G) \cap \mathcal{L}_m(G)] \cup [\ \mathcal{L}(S_2/G)\ ) \cap \mathcal{L}_m(G)\ ] &(7)\\
&= \mathcal{L}_m(S_1/G) \cup \mathcal{L}_m(S_2/G) &(8)
\end{aligned}$$

Remark 5 (Supervisor disjunction) It is worth noting that, without the assumption in Eq. (3), the closed-loop language under disjunction of supervisors may not be the union of their respective languages. As an example, consider again plant G_3 of Fig. 5, with $L_{am} := \{\varepsilon, c_1\}$. Consider supervisors S_1 and S_2 where S_1 always disables c_1 and enables c_2, whereas S_2 always disables c_2 and enables c_1. Both S_1 and S_2 are non-blocking for G_3, because all states in the closed-loop system are accepting. Moreover, both S_1 and S_2 are safe w.r.t. the above L_{am}. Indeed, S_2 always disables c_2, while S_1, by disabling c_1, prevents G_3 from reaching state x_1, thus indirectly preventing c_2. However, $S_1 \cup S_2$ is not safe, since it allows both c_1 and c_2 at any state.

The following result establishes a key property of supervisor disjunction.

Theorem 5 *Let G be a DES plant and let $L_{am} \subseteq \mathcal{L}_m(G)$ be the admissible marked language. If S_1, S_2 are two non-blocking and safe supervisors that are G-matched, then $S_1 \cup S_2$ is also a non-blocking and safe supervisor, and it is G-matched.*

Proof We have that $\overline{\mathcal{L}_m(S_i/G)} = \mathcal{L}(S_i/G)$ and $\mathcal{L}_m(S_i/G) \subseteq L_{am}$, for $i = 1, 2$.

Safety: Clearly,

$$\mathcal{L}_m(S_{1\cup 2}/G) = \mathcal{L}_m(S_1/G) \cup \mathcal{L}_m(S_2/G) \subseteq L_{am}$$

and thus $S_1 \cup S_2$ is a safe supervisor.

Non-blockingness: Since prefix-closure can be distributed over union (the proof of this fact is straightforward), we have that

$$\begin{aligned}\overline{\mathcal{L}_m(S_{1\cup 2}/G)} &= \overline{\mathcal{L}_m(S_1/G) \cup \mathcal{L}_m(S_2/G)} &(9)\\ &= \overline{\mathcal{L}_m(S_1/G)} \cup \overline{\mathcal{L}_m(S_2/G)} &(10)\\ &= \mathcal{L}(S_1/G) \cup \mathcal{L}(S_2/G) &(11)\\ &= \mathcal{L}(S_{1\cup 2}/G). &(12)\end{aligned}$$

which proves that $S_1 \cup S_2$ is a non-blocking supervisor.

Finally, it is clear that $S_{1\cup 2}$ is G-matched as it inherits this property from the definitions of S_1 and S_2 outside of $\mathcal{L}(S_{1\cup 2}/G)$. □

Corollary 3 *Theorem 5 holds for infinite disjunctions of supervisors.*

Proof All steps in the proof of Theorem 5 hold for an arbitrary number of disjunctions. □

The hypothesis of Theorem 1 is that there exists at least one non-blocking supervisor for G that is safe w.r.t. L_{am}. If there exists a single supervisor with these properties, then it is necessarily the unique desired S_{mpnb}, as no other safe non-blocking supervisor exists. Let us assume then that there are several safe and non-blocking supervisors. If a supervisor S is not G-matched, then we can always make it G-matched without changing $\mathcal{L}(S/G)$ or $\mathcal{L}_m(S/G)$. By taking the disjunction of *all* G-matched non-blocking supervisors for G that are safe w.r.t. L_{am}, we obtain a unique G-matched supervisor that is also safe and non-blocking by Corollary 3; let us denote it by S^G_{disj}. Then $\mathcal{L}_m(S^G_{disj})$ contains all the sublanguages of L_{am} that can be achieved by any safe and non-blocking supervisor. Otherwise, if the sublanguage of L_{am} achieved by one supervisor is not contained in $\mathcal{L}_m(S^G_{disj})$, then the G-matched version of that supervisor would not have been added in the disjunction of all G-matched safe and non-blocking supervisors, a contradiction.

Once we have the unique maximally-permissive closed-loop behavior $\mathcal{L}_m(S^G_{disj})$, we take S_{mpnb} to be the unique maximally permissive supervisor that achieves it. To obtain S_{mpnb}, we simply add to S^G_{disj} all infeasible (in G) controllable events for strings in $\mathcal{L}(S^G_{disj}/G)$ and all controllable events for strings in $E^* \setminus \mathcal{L}(S^G_{disj}/G)$. Since $\mathcal{L}(S_{mpnb}/G) = \mathcal{L}(S^G_{disj}/G)$ and $\mathcal{L}_m(S_{mpnb}/G) = \mathcal{L}_m(S^G_{disj}/G)$, then S_{mpnb} is non-blocking for G and safe w.r.t. L_{am}. S_{mpnb} is not G-matched anymore, but this is of no consequence in the later developments in the paper.

This completes the proof of Theorem 1.

B Proof of theorem 2

First, we state and prove the following lemma.

Lemma 5 *Let $G = (X, x_0, X_m, E, \delta)$, $L_{am} \subseteq \mathcal{L}_m(G)$, and assume that L_{am} is $\mathcal{L}_m(G)$-closed. Let A be a complete DFA such that $\mathcal{L}_m(A) = L_{am}$. Let S be a supervisor for G, and therefore also for $G \times A$. Then, the following hold:*

1. *If S is non-blocking for plant $G \times A$, then S is non-blocking for plant G.*
2. *If S is non-blocking for plant $G \times A$, then S is safe for plant G w.r.t. L_{am}.*

3. *If S is non-blocking for plant G and safe for plant G w.r.t. L_{am}, then S is non-blocking for plant $G \times A$.*
4. *S is safe for plant $G \times A$ w.r.t. $\mathcal{L}_m(G \times A)$.*

Proof 1. To show that S is non-blocking for G, we need to show $\mathcal{L}(S/G) \subseteq \overline{\mathcal{L}_m(S/G)}$. Let $\sigma \in \mathcal{L}(S/G)$. We need to find σ' such that $\sigma \cdot \sigma' \in \mathcal{L}_m(S/G)$. We know that $\sigma \in \mathcal{L}(S/G \times A)$ because A is complete. Also, S is non-blocking for $G \times A$, thus $\sigma \in \overline{\mathcal{L}_m(S/G \times A)}$. Therefore there exists σ' such that $\sigma \cdot \sigma' \in \mathcal{L}_m(S/G \times A)$. By definition of the marked states of $G \times A$, both G and A accept $\sigma \cdot \sigma'$. Therefore, $\sigma \cdot \sigma' \in \mathcal{L}_m(S/G)$.

2. To show that S is safe for G w.r.t. L_{am}, we need to show $\mathcal{L}_m(S/G) \subseteq L_{am}$. Let $\sigma \in \mathcal{L}_m(S/G)$. We need to show $\sigma \in L_{am}$. $\sigma \in \mathcal{L}_m(S/G)$ implies $\sigma \in \mathcal{L}(S/G)$, and therefore $\sigma \in \mathcal{L}(S/G \times A)$ because A is complete. Consider the (unique) run of $G \times A$ on σ. We claim that this run ends on a product state that is marked for both G and A. $\sigma \in \mathcal{L}_m(S/G)$, therefore the product state must indeed be marked for G. We show that $\sigma \in \mathcal{L}_m(A)$ which implies that the product state is also marked for A. Since S is non-blocking for $G \times A$, there exists σ' such that $\sigma \cdot \sigma' \in \mathcal{L}_m(S/G \times A)$. But this means $\sigma \cdot \sigma' \in \mathcal{L}_m(A)$, which implies $\sigma \in \overline{\mathcal{L}_m(A)}$. Also, $\sigma \in \mathcal{L}_m(S/G)$ implies $\sigma \in \mathcal{L}_m(G)$. Therefore, we have $\sigma \in \mathcal{L}_m(G) \cap \overline{\mathcal{L}_m(A)}$.

   Since $\mathcal{L}_m(A)$ (i.e., L_{am}) is $\mathcal{L}_m(G)$-closed, $\sigma \in \mathcal{L}_m(A)$.

3. To show that S is non-blocking for plant $G \times A$, we need to show $\mathcal{L}(S/G \times A) \subseteq \overline{\mathcal{L}_m(S/G \times A)}$. Let $\sigma \in \mathcal{L}(S/G \times A)$. We need to find σ' such that $\sigma \cdot \sigma' \in \mathcal{L}_m(S/G \times A)$. $\sigma \in \mathcal{L}(S/G \times A)$ implies $\sigma \in \mathcal{L}(S/G)$ and $\sigma \in \mathcal{L}(A)$. Since S is non-blocking for G, there exists σ' such that $\sigma \cdot \sigma' \in \mathcal{L}_m(S/G)$. Since $\mathcal{L}_m(S/G) \subseteq L_{am} = \mathcal{L}_m(A)$, $\sigma \cdot \sigma' \in \mathcal{L}_m(A)$. $\sigma \cdot \sigma' \in \mathcal{L}_m(S/G)$ and $\sigma \cdot \sigma' \in \mathcal{L}_m(A)$ implies $\sigma \cdot \sigma' \in \mathcal{L}_m(S/G \times A)$. Therefore $\sigma \in \overline{\mathcal{L}_m(S/G \times A)}$.

4. Trivially, since safe w.r.t. $\mathcal{L}_m(G \times A)$ means $\mathcal{L}_m(S/G \times A) \subseteq \mathcal{L}_m(G \times A)$, which holds for any S.

□

We can now present the proof of Theorem 2.

Proof 2 $\Rightarrow$ 1 : We need to show that S is non-blocking for G, safe w.r.t. L_{am}, and maximally-permissive. Non-blockingness follows from Lemma 5, part 1. Safety follows from Lemma 5, part 2.

To show that S is maximally-permissive in G, suppose there exists a non-blocking and safe w.r.t. L_{am} supervisor S' which is strictly more permissive than S. By Lemma 5, parts 3 and 4, S' is non-blocking for $G \times A$ and safe for $G \times A$ w.r.t. $\mathcal{L}_m(G \times A)$. The fact that S' is strictly more permissive than S in G also means that S' is strictly more permissive than S in $G \times A$. This contradicts the hypothesis that S is maximally-permissive in $G \times A$.

1 $\Rightarrow$ 2 : We need to show that S is non-blocking for $G \times A$, safe for $G \times A$ w.r.t. $\mathcal{L}_m(G \times A)$, and maximally-permissive. Non-blockingness follows from Lemma 5, part 3. Safety follows from Lemma 5, part 4.

To show that S is maximally-permissive in $G \times A$, suppose there exists a non-blocking supervisor S' for $G \times A$ (and also trivially safe w.r.t. $\mathcal{L}_m(G \times A)$) which is strictly more permissive than S.

By Lemma 5, parts 1 and 2, S' is non-blocking for G and safe for G w.r.t. L_{am}. The fact that S' is strictly more permissive than S in $G \times A$ also means that S' is strictly more permissive than S in G. This contradicts the hypothesis that S is maximally-permissive in G. □

C An algorithm for SSCP

For the sake of completeness of this paper, a simple algorithm to solve SSCP is presented below. The algorithm starts by labeling as *Blocking* all states that cannot reach a marked state (note that this generally requires reachability analysis). Then the algorithm iterates, repeatedly labeling more states as *Blocking*, until no more can be labeled. A state is labeled during this iteration, if either it has an uncontrollable successor already labeled, or all its successors are already labeled. At the end, if the initial state is labeled *Blocking* then no supervisor exists. Otherwise, a state-based supervisor can easily be constructed by avoiding all controllable transitions leading to *Blocking* states.

Algorithm Solve SSCP.

```
Input: DES $G = (X, x_0, X_m, E, \delta)$, with $E = E_c \cup E_{uc}$.
Output: A maximally-permissive non-blocking supervisor $S$ for $G$, if one exists, "no" otherwise.
$Blocking := \{x \in X - X_m \mid \nexists$ path from $x$ to any state in $X_m\}$;
repeat
    $StatesWithUncontrollablyBlockingSuccs := \{x \in X - X_m \mid \exists u \in E_{uc} : \delta(x,u) \in Blocking\}$;
    $NewDeadlocks := \{x \in X - X_m \mid \forall e \in E : \delta(x,e) \in Blocking$ or $\delta(x,e)$ is undefined$\}$;
    $Blocking := Blocking \cup StatesWithUncontrollablyBlockingSuccs \cup NewDeadlocks$;
until set $Blocking$ does not change;
if ($x_0 \in Blocking$) then return "no supervisor exists";
else
    let for $x \in X$, $S(x) := E_{uc} \cup \begin{cases} E_c & \text{if } x \in Blocking \\ \{c \in E_c \mid \delta(x,c) \notin Blocking\} & \text{otherwise} \end{cases}$;
    return $S$;
```

The above algorithm essentially computes a fixpoint of a function that takes a set of blocking states and returns those states that have an uncontrollable transition to a blocking successor or all of whose successor states are blocking. The fixpoint evaluation starts from the set of states that do not have a path to a marked state. The algorithm is furthermore an adaptation of the standard algorithm in Wonham and Ramadge (1987) for solving BSCP-NB to the special case of SSCP. Examples of the application of the standard algorithm for solving BSCP-NB can be found in Wonham (2015), Cassandras and Lafortune (2008), for instance.

As a simple example of how the algorithm operates, consider the plant G_1 of Figure 3. After initialization, *Blocking* contains a single state, namely x_2. (Note that x_1 has no path to itself, but is not considered blocking because it is marked.) After executing the body of the repeat loop no more states are added to the set *Blocking*. Indeed, no state has an uncontrollable transition to x_2, therefore the set *StatesWithUncontrollablyBlockingSuccs* turns out to be empty. The set *NewDeadlocks* also turns out to be empty because there is no state whose only successors, if any, are blocking, i.e., x_2 (note that, again, the marked state x_1 is exempted). Since *Blocking* does not change, the loop is exited after one iteration. The initial state is not blocking, therefore a supervisor exists and is defined as follows: $S(x_0) = \{u, c_1\}$, $S(x_1) = S(x_2) = \{u, c_1, c_2\}$, and $S(x_3) = \{c_2\}$. The computed supervisor corresponds to supervisor S_2 shown in Fig. 3.

References

Abadi M, Lamport L, Wolper P (1989) Realizable and unrealizable concurrent program specifications. In: Proceedings of the 25th International Colloq. on Automata, Languages, and Programming, volume 372 of Lecture Notes in Computer Science. Springer, pp 1–17

Asarin E, Maler O, Pnueli A (1995) Symbolic controller synthesis for discrete and timed systems. In: Hybrid Systems II

Arnold A, Vincent A, Walukiewicz I (2003) Games for synthesis of controllers with partial observation. Theor Comput Sci 303(1):7–34

Büchi JR, Landweber LH (1969) Solving sequential conditions by finite-state strategies. Trans. AMS 138:295–311

Barrett G, Lafortune S (1998) On the synthesis of communicating controllers with decentralized information structures for discrete-event systems. In: IEEE Conference on Decision and Control

Clarke EM, Emerson EA (1981) Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proceedings Workshop on Logic of Programs, volume 131 of Lecture Notes in Computer Science, pp 52–71. Springer

Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8(2):244–263

Clarke E, Grumberg O, Peled D (2000) Model Checking. MIT Press

Cassez F, Henzinger T, Raskin J-F (2002) A Comparison of Control Problems for Timed and Hybrid Systems. In: HSCC'02, volume 2289 of LNCS. Springer-Verlag

Church A (1957) Applicaton of recursive arithmetics to the problem of circuit synthesis. In: Summaries of Talks Presented at The Summer Institute for Symbolic Logic, 3–50. Communications Research Division, Institute for Defense Analysis

Church A (1963) Logic, arithmetic and automata. In: Proceedings of the International Congress of Mathematics

Cassandras CG, Lafortune S (2008) Introduction to Discrete Event Systems, 2nd. Springer, Boston

Dill DL (1989) Trace theory for automatic hierarchical verification of speed independent circuits. MIT Press

de Queiroz MH, Cury JER, Wonham WM (2005) Multitasking supervisory control of discrete-event systems. Discrete Event Dynamic Systems. Theory Appl 15(4):375–395

Emerson EA, Clarke EM (1982) Using branching time logic to synthesize synchronization skeletons. Sci Comput Program 2:241–266

Emerson EA, Halpern JY (1986) Sometimes and Not Never revisited: on branching versus linear time temporal logic. J. ACM 33(1):151–178

Ehlers R (2013) Symmetric and Efficient Synthesis. PhD thesis, Universität des Saarlandes

Ehlers R, Lafortune S, Tripakis S, Vardi M (2014) Bridging the gap between supervisory control and reactive synthesis: Case of full observation and centralized control. In: Proceedings of the 12th International Workshop on Discrete Event Systems (WODES 2014)

Francez N (1992) Program verification. Int. Computer Science. Addison-Weflay

Green CC (1969) Application of theorem proving to problem solving. In: 1st International Joint Conference on Artificial Intelligence, pp 219–240

Henzinger T, Kopke P (1997) Discrete-time control for rectangular hybrid automata. In: ICALP '97

Harel D, Marelly R (2003) Come, Let's Play. Springer

Harel D, Pnueli A (1985) On the development of reactive systems. In: Apt K (ed) Logics and Models of Concurrent Systems, volume F-13 of NATO Advanced Summer Institutes. Springer, pp 477–498

Hoffmann G, Wong Toi H (1992) Symbolic synthesis of supervisory controllers. In: American Control Conference

Jackson D (2009) A direct path to dependable software. Commun. ACM 52(4):78–88

Jiang S, Kumar R (2006) Supervisory Control of Discrete Event Systems with CTL* Temporal Logic Specifications. SIAM J Control Optim 44(6):2079–2103

Kress-Gazit H, Fainekos GE, Pappas GJ (2007) Where's waldo? Sensor-based temporal logic motion planning. In: IEEE International Conference on Robotics and Automation, ICRA, pp 3116–3121

Kumar R, Garg VK (1995) Modeling and control of logical discrete event systems. Kluwer Academic Publishers

Kumar R, Garg V, Marcus SI (1992) On supervisory control of sequential behaviors. IEEE Trans Autom Control 37(12):1978–1985

Kupferman O, Madhusudan P, Thiagarajan PS, Vardi MY (2000) Open systems in reactive environments: Control and synthesis. In: 11th International Conference on Concurrency Theory, CONCUR'00. Springer, pp 92–107

Komenda J, Masopust T, van Schuppen JH (2012) Supervisory control synthesis of discrete-event systems using a coordination scheme. Automatica 48(2):247–254

Kupferman O, Vardi M (1996) Module checking. In: Alur R, Henzinger T (eds) Computer Aided Verification, volume 1102 of LNCS. Springer, pp 75–86

Kupferman O, Vardi M (1999) Church's problem revisited. Bull Symb Log 5(2)

Kupferman O, Vardi MY (2000) Synthesis with incomplete information. Advances in Temporal Logic, pp 109–127. Kluwer Academic Publishers

Lin F (1993) Analysis and synthesis of discrete event systems using temporal logic. Control Theory Adv Technol 9:341–350

Lichtenstein O, Pnueli A (1985) Checking that finite state concurrent programs satisfy their linear specification. In: Proceedings 12th ACM Symposium on Principles of Programming Languages, pp 97–107

Lamouchi H, Thistle J (2000) Effective control synthesis for DES under partial observations. In: 39th IEEE Conference on Decision and Control, pp 22–28

Lustig Y, Vardi M (2009) Synthesis from component libraries, Foundations of Software Science and Computational Structures, pp 395–409

Madhusudan P (2001) Control and Synthesis of Open Reactive Systems. PhD thesis, University of Madras

Maler O, Pnueli A, Sifakis J (1995) On the synthesis of discrete controllers for timed systems. In: STACS '95

Manna Z, Wolper P (1984) Synthesis of communicating processes from temporal logic specifications. ACM TOPLAS 6(1)

Overkamp A, van Schuppen JH (2000) Maximal solutions in decentralized supervisory control. SIAM J Control Optim 39(2):492–511

Pnueli A (1977) The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, pp 46–57

Piterman N, Pnueli A, Saar Y (2006) Synthesis of reactive(1) designs. In: Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation, volume 3855 of Lecture Notes in Computer Science. Springer, pp 364–380

Pnueli A, Rosner R (1989a) On the synthesis of a reactive module. In: ACM Symposium POPL

Pnueli A, Rosner R (1989b) On the synthesis of an asynchronous reactive module. In: Proceedings of the 16th International Colloq. on Automata, Languages, and Programming, volume 372 of Lecture Notes in Computer Science. Springer, pp 652–671

Pnueli A, Rosner R (1990) Distributed reactive systems are hard to synthesize. In: Proceedings of the 31th IEEE Symposium on Foundations of Computer Science, pp 746–757

Pinchinat S, Riedweg S (2005) You can always compute maximally permissive controllers under partial observation when they exist. In: Proceedings of the 2005 American Control Conference, pp 2287–2292

Queille JP, Sifakis J (1982) Specification and verification of concurrent systems in Cesar. In: Proceedings of the 8th ACM Symposium on Principles of Programming Languages, volume 137 of Lecture Notes in Computer Science. Springer, pp 337–351

Rabin MO (1972) Automata on infinite objects and Church's problem. Amer. Mathematical Society

Ramadge PJ (1989) Some tractable supervisory control problems for discrete-event systems modeled by Büchi automata. IEEE Trans Autom Control 34(1):10–19

Riedweg S, Pinchinat S (2003) Quantified mu-calculus for control synthesis. In: Rovan B, Vojas P (eds) Mathematical Foundations of Computer Science 2003, volume 2747 of Lecture Notes in Computer Science. Springer, Berlin, pp 642–651

Riedweg S, Pinchinat S (2004) Maximally permissive controllers in all contexts. In: Proceedings of the 2004 International Workshop on Discrete Event Systems

Ricker SL, Rudie K (2000) Know means no: Incorporating knowledge into discrete-event control systems. IEEE Trans Autom Control 45(9):1656–1668

Ramadge P, Wonham W (1987) Supervisory control of a class of discrete event processes. SIAM J Control Optim 25(1):206–230

Ramadge P, Wonham W (1989) The control of discrete event systems, In: Proceedings of the IEEE

Rudie K, Wonham W (1992) Think globally, act locally: Decentralized supervisory control. IEEE Trans Autom Control:37

Seatzu C, Silva M, van Schuppen J (eds) (2013) Control of Discrete Event Systems. Automata and Petri Net Perspectives. Springer

Thistle JG (1995) On control of systems modelled as deterministic Rabin automata. Discrete Event Dynamic Systems 5(4):357–381

Thistle JG (1996) Supervisory control of discrete event systems. Mathl Comput Modelling 23(11/12):25–53

Thistle JG, Malhamé RP (1998) Control of omega-automata under state fairness assumptions. Syst Control Lett:33

Tripakis S (2004) Undecidable Problems of Decentralized Observation and Control on Regular Languages. Inf Process Lett 90(1):21–28

Thistle JG, Wonham WM (1986) Control problems in a temporal logic framework. Int J Control 44(4):943–976

Thistle J, Wonham W (1994a) Control of infinite behavior of finite automata. SIAM J Control Optim 32(4):1075–1097

Thistle J, Wonham W (1994b) Supervision of infinite behavior of discrete-event systems. SIAM J Control Optim 32(4):1098–1113

Vardi MY (1995) An automata-theoretic approach to fair realizability and synthesis. In: Proceedings of the 7th International Conference on Computer Aided Verification, volume 939 of Lecture Notes in Computer Science. Springer, pp 267–292

van Hulst AC, Reniers MA, Fokkink WJ (2014) Maximal synthesis for Hennessy-Milner logic with the box modality. In: Proceedings of the 2014 International Workshop on Discrete Event Systems

Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification. In: Proceedings of the 1st IEEE Symposium on Logic in Computer Science, pp 332–344

Wonham WM (2015) Supervisory Control of Discrete Event Systems. Available on the author's website

Wonham W, Ramadge P (1987) On the supremal controllable sublanguage of a given language. SIAM J. Control Optim. 25(3):637–659

Wong-Toi H, Dill DL (1991) Synthesizing processes and schedulers from temporal specifications. In: Clarke EM, Kurshan RP (eds) Proc. 2nd Int. Conf. on Computer Aided Verification, volume 3 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp 177–186. AMS

Rüdiger Ehlers is Junior Research Group Leader at the University of Bremen, cooperating with the German Research Center for Artificial Intelligence (DFKI). He received a Diploma in Computer Science from the Technical University of Dortmund, an M.Sc. in Mathematics & the Foundations of Computer Science from the University of Oxford, UK, and a Dr. in Computer Science from Saarland University, Germany. He was shared researcher between the University of California at Berkeley and Cornell University, USA, from 2012-2013, and a postdoc at the University of Kassel, Germany, from 2013 to 2014.

His research deals with methods to synthesize and verify cyber-physical systems, including efficient symbolic reasoning engines for these goals.

Stéphane Lafortune received the B.Eng degree from Ecole Polytechnique de Montréal in 1980, the M.Eng degree from McGill University in 1982, and the Ph.D degree from the University of California at Berkeley in 1986, all in electrical engineering. Since September 1986, he has been with the University of Michigan, Ann Arbor, where he is Professor of Electrical Engineering and Computer Science. Dr. Lafortune is a Fellow of the IEEE (1999). He received the Presidential Young Investigator Award from the National Science Foundation in 1990 and the George S. Axelby Outstanding Paper Award from the Control Systems Society of the IEEE in 1994 (for a paper co-authored with S.-L. Chung and F. Lin) and in 2001 (for a paper co-authored with G. Barrett). Dr. Lafortune's research interests are in discrete event systems and include multiple problem domains: modeling, diagnosis, control, optimization, and applications to computer and software systems. He is the lead developer of the software package UMDES and co-developer of DESUMA with L. Ricker. He co-authored, with C. Cassandras, the textbook *Introduction to Discrete Event Systems - Second Edition* (Springer, 2008). Dr. Lafortune is Editor-in-Chief of the Journal of Discrete Event Dynamic Systems: Theory and Applications.

Stavros Tripakis is an Associate Professor at Aalto University, and an Adjunct Associate Professor at the University of California, Berkeley. He received a Ph.D. degree in Computer Science in 1998 at the Verimag Laboratory, Joseph Fourier University, Grenoble, France. He was a Postdoc at UC Berkeley from 1999 to 2001, a CNRS Research Scientist at Verimag from 2001 to 2006, and a Research Scientist at Cadence Research Labs, Berkeley, from 2006 to 2008. His research interests include formal methods, computer-aided system design, and cyber-physical systems. Dr. Tripakis was co-Chair of the 10th ACM & IEEE Conference on Embedded Software (EMSOFT 2010), and Secretary/Treasurer (2009-2011) and Vice-Chair (2011-2013) of ACM SIGBED. His h-index is 40 (Google Scholar).

Moshe Y. Vardi is the George Distinguished Service Professor in Computational Engineering and Director of the Ken Kennedy Institute for Information Technology at Rice University. He is the recipient of three IBM Outstanding Innovation Awards, the ACM SIGACT Gödel Prize, the ACM Kanellakis Award, the ACM SIGMOD Codd Award, the Blaise Pascal Medal, the IEEE Computer Society Goode Award, the EATCS Distinguished Achievements Award, and the Southeastern Universities Research Association's Distinguished Scientist Award. He is the author and co-author of over 500 papers, as well as two books: "Reasoning about Knowledge" and "Finite Model Theory and Its Applications". He is a Fellow of the Association for Computing Machinery, the American Association for Artificial Intelligence, the American Association for the Advancement of Science, the European Association for Theoretical Computer Science, the Institute for Electrical and Electronic Engineers, and the Society for Industrial and Applied Mathematics. He is a member of the US National Academy of Engineering and National Academy of Science, the American Academy of Arts and Science, the European Academy of Science, and Academia Europaea. He holds honorary doctorates from the Saarland University in Germany, Orleans University in France, and UFRGS in Brazil. He is the Editor-in-Chief of the Communications of the ACM.