Compositional Supervisory Control via Reactive Synthesis and Automated Planning

Daniel Ciolek⋆, Victor Braberman⋆, Nicolás D'Ippolito⋆, Sebastián Uchitel⋆ and Sebastián Sardiña†

***Abstract*—We show how reactive synthesis and automated planning can be leveraged effectively to find non-maximal solutions to deterministic supervisory control problems of discrete event systems. To do so, we propose efficient translations of the supervisory control problem into the reactive synthesis and planning frameworks. Notably, our translation methods capture the compositional and reactive nature of control specifications, avoiding a potential exponential explosion found in alternative translation approaches. Additionally, we report on experimental results comparing the efficacy of different tools from the three disciplines, for a particular supervisory control benchmark.**

***Index Terms*—Supervisory Control, Reactive Synthesis, Automated Planning**

I. Introduction

SUPERVISORY Control [**?**], Reactive Synthesis [**?**] and Automated Planning [**?**] are three disciplines that look for an orderly combination of events/signals/actions guaranteeing a given goal specification. Arising from different communities, they consider distinct perspectives on representational and computational aspects. In this work, we contribute to recent efforts in relating these three fields [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. Concretely, we show how to leverage on reactive synthesis and automated planning techniques to tackle modular deterministic supervisory control problems. To that end we provide provably correct compilations and an empirical comparison between tools from the three areas.

In supervisory control, Discrete Event Systems (DES) are expressed by relying on a modular approach based on the *parallel composition* of multiple interacting components [**?**], referred to as the plant. Supervisory control aims at controlling such DES to achieve certain guarantees. This is done by deploying a so-called "supervisor" that disables controllable events while monitoring uncontrollable events. Traditional supervisory control techniques look for maximally permissive supervisors, a task that has been argued to require a prohibitive computational cost [**?**], [**?**]. In this work we focus on a subset of solutions for deterministic supervisory control problems that do not require maximality.

In the field of reactive synthesis a problem specification describes the desired behavior of the controller at the interface level. Such specifications are usually expressed in a temporal logic [**?**], such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). Controllers are seen as open dynamical systems, in the sense that they have input and output signals which affect the system dynamics. In this setting, complex systems are usually specified as the conjunction of simpler sub-modules.

Automated Planning (or planning for short) stems from a different tradition, namely, Knowledge Representation within Artificial Intelligence. A planning problem is specified by describing the preconditions and effects of actions in the domain, together with the goal to be achieved. Based on reasoning about action languages, effective techniques have been developed over the years [**?**]. However, the work in planning, unlike that in supervisory control and reactive synthesis, has been mainly oriented towards non-reactive environments. Still, advances in planning have led to tackle non-deterministic problems which we rely upon to model reactivity.

Interestingly, the three disciplines share an important characteristic, which is that input specifications are given via *compact descriptions*. That is, the semantics of the problems to be solved are based on finite state transitions systems that are often *exponential* with respect to the number of individual components. This state explosion has been dealt with in the three fields relying on different approaches. For instance, in supervisory control compositional analyses (e.g. [**?**]) are performed on individual components allowing for a (potentially) efficient merging procedure that satisfies the problem requirements; in reactive synthesis symbolic representations (e.g. [**?**]) are used to encode the problem in a "compressed" structure which can be processed efficiently; and in automated planning informed search procedures (e.g. [**?**]) are used to find a solution by exploring (hopefully) only a reduced portion of the state space.

All the above problems are known to be computationally challenging. Supervisory control is PTIME with respect to the size of the overall plant [**?**], but since we consider a compositional description of the plant, the problem is ultimately EXPTIME with respect to the number of individual components. Reactive synthesis is 2EXPTIME-complete [**?**], yet by restricting the input (e.g., relying on CTL or GR(1) requirements [**?**], [**?**]) the second exponential explosion can often be circumvented. Similarly, classical deterministic planning is PSPACE-complete with respect to the succinct planning specification, the addition of non-deterministic actions makes the problem EXPTIME-complete [**?**]. From a theoretical point of view worst case complexity may seem intractable, yet in all three disciplines algorithms targeting applicability exist and provide huge benefits in practice.

The main contribution of this paper is the compilation from a compositional supervisory control problem into (a) a reactive

⋆ Departamento de Computación, UBA, Argentina.
† School of Computer Science and IT, RMIT, Australia.
Manuscript received DATE; revised DATE.
This work was partially supported by grants ANPCYT PICT 2014-1656, ANPCYTPICT 2015-3638, ANPCYT PICT 2015-1718, UBACYT 036, UBACYT 384, CONICET PIP 2014/16 N°11220130100688CO.

synthesis problem and (b) a (non-deterministic) planning task. Remarkably, the compilation avoids the construction of the exponential semantic model by explicitly modeling parallel composition and synchronization among components. That is to say, the translations begin with a compact supervisory control specification and produce compact reactive synthesis and planning specifications in polynomial time. Furthermore, we report on the first experimental evaluation comparing publicly available software tools from the three disciplines over a supervisory control benchmark. The results show that the performance of state-of-the-art planning and reactive synthesis techniques – under our encoding – rival that of native supervisory control tools.

II. Related Work

To the best of our knowledge this work is the first attempt to reduce a supervisory control problem into planning and reactive synthesis *exploiting the compositional aspects of supervisory control specifications*. However, there has been great interest in relating these fields and in this section we discuss approaches that share characteristics with our work.

In [**?**] and [**?**] reductions from supervisory control to reactive synthesis are presented. The approaches are *monolithic*, in the sense that they consider as input to the supervisory control problem a single automaton (the result of the composition of multiple interacting components). Thus, despite being formally correct, these approaches may not be practical for handling complex compositional specifications since computing the composition may result in an exponential blowup.

In [**?**] and [**?**] reductions from planning to supervisory control are presented. However, this line of work focuses on characterizing fairness assumptions and relies on a *monolithic* reduction. Thus, it can also incur in an exponential cost.

In [**?**] a compilation from reactive synthesis into planning is presented. Interestingly, the compilation passes through an intermediate representation based on a Finite State Machine (FSM), which resembles the formalism used in supervisory control. Similarly, in [**?**] a compilation from reactive synthesis into planning that passes through a non-deterministic Büchi automaton, instead of an FSM, is presented together with a number of optimizations. Still, both approaches rely on *monolithic* FSM/automata, and hence do not tackle the challenges of the compositional case.

In [**?**] a reduction from ConGolog (a logical programming language for concurrent agents) to situation calculus (a common logical interpretation for planning) is presented. This translation shares some characteristics with ours, namely the schematization of different phases in the execution semantics of concurrent models. However, their translation builds a Petri-Net accommodating a compositional approach by relying on multiple-token semantics, whereas we explicitly encode the rules of parallel composition.

In [**?**] a framework for the composition of "behaviors" is presented. The technique allows computing a supremal controller from smaller controllers for sub-modules, such that it remains correct under the presence of exogenous events. Despite taking as input a compositional specification, the synthesis approach is *monolithic* (i.e., it works over the composition of behaviors). We believe our translations can provide insights on how to extend that work to exploit the compositional nature of the problem.

Temporal logics – such as LTL and CTL – have been adopted in both, supervisory control and planning, as a means for describing declarative liveness goals [**?**], [**?**], [**?**]. Some approaches reduce problems to the reactive synthesis framework, while others explicitly encode the liveness constraints in their own framework. Despite methodological similarities with our work, none of these approaches consider compositional representations as the ones typically used in supervisory control, nor compare results with tools from this field.

The existing body of work highlights the similarities between the disciplines and the interest of the different communities in relating the fields. Despite the fact that traditional supervisory control takes a *monolithic* approach to the synthesis of a maximally permissive supervisor, there is a growing interest in *compositional* [**?**], [**?**] and *non-maximal* approaches [**?**], [**?**]. The combination of the two brings supervisory control close to planning and reactive synthesis, making the question on their relationship relevant. Thus, in this paper we build in this direction by providing provably correct translations from non-maximal compositional supervisory control problems that do not incur in an exponential blowup.

III. Background

A. Supervisory Control

Supervisory control theory (SC) [**?**], [**?**] aims at controlling a discrete system with an interaction model based on events, by enabling/disabling a distinguished set of controllable events in order to guarantee a desired closed loop behavior. For modeling and computational reasons [**?**], [**?**] the system, commonly referred to as the *plant*, is generally expressed modularly as the *parallel composition* of its components/sub-systems. Since SC is rooted in classical formal languages, the plant and its components are seen as generators of the languages of strings of events characterizing their processes, and are often modeled as *deterministic automata*.

Definition 1 (Deterministic Automaton)**.** A *deterministic automaton* is a tuple $T = (S_T, A_T, \rightarrow_T, t_0, M_T)$, where S_T is a *finite set of states*; A_T is the automaton *event set*; $\rightarrow_T \subseteq (S_T \times A_T \times S_T)$ is a *transition function*; $t_0 \in S_T$ is the *initial state*; and $M_T \subseteq S_T$ is a set of *marked states*.

We denote $(t, \ell, t') \in \rightarrow_T$ by $t \xrightarrow{\ell}_T t'$ and call it a *step*. In turn, a *run* on a word $w = \ell_0, \ldots, \ell_k$ in T is a sequence of steps $t_0 \xrightarrow{\ell_0}_T t_1 \xrightarrow{\ell_1}_T \cdots \xrightarrow{\ell_k}_T t_{k+1}$ and is denoted $t_0 \overset{w}{\dashrightarrow}_T t_{k+1}$.

As customary, an automaton T defines a language, in our case, over its event set A_T. The language *generated* by T is defined as $\mathcal{L}(T) = \{w \in A_T^* \mid t_0 \overset{w}{\dashrightarrow}_T t'\}$, where A_T^* is the set of finite words of elements of A_T. Marked states (i.e., states in M_T) are used to indicate the termination of a task. Thus, we additionally consider the language *marked* (or *accepted*) by T (denoted $\mathcal{L}_m(T)$) as the set of strings producing runs that end on a marked state. Formally,

$$\mathcal{L}(T) = \{w \in A_T^* \mid t_0 \overset{w}{\dashrightarrow}_T t', t' \in M_T\}$$

To compose the joint behavior of two components, we define the parallel composition of their corresponding automata, defined, broadly, as their synchronous product.

Definition 2 (Parallel Composition)**.** The *parallel composition* ($\|$) of two automata T and Q is an automaton defined as $T\|Q = (S_T \times S_Q, A_T \cup A_Q, \rightarrow_{T\|Q}, \langle t_0, q_0\rangle, M_T \times M_Q)$, where $\rightarrow_{T\|Q}$ is the smallest relation satisfying the following rules:

$$\frac{t \xrightarrow{\ell}_T t'}{\langle t,q\rangle \xrightarrow{\ell}_{T\|Q} \langle t',q\rangle}\ \ell \in A_T \setminus A_Q \qquad \frac{q \xrightarrow{\ell}_Q q'}{\langle t,q\rangle \xrightarrow{\ell}_{T\|Q} \langle t,q'\rangle}\ \ell \in A_Q \setminus A_T$$

$$\frac{t \xrightarrow{\ell}_T t' \quad q \xrightarrow{\ell}_Q q'}{\langle t,q\rangle \xrightarrow{\ell}_{T\|Q} \langle t',q'\rangle}\ \ell \in A_T \cap A_Q$$

We make some important observations on the parallel composition. First, $\|$ is an associative symmetric operator, so we shall write $T_0\| \ldots \|T_n$ to denote the composition of multiple automata/components. Second, the third rule realizes synchronization between the components by enforcing the *synchronous execution over the shared events*. Third, the language accepted by the composition contains the strings that reach marked states in *all* components simultaneously. Finally, the number of states in the composition may grow exponentially with the number automata, thus imposing computational challenges when the overall plant is the result of integrating a large number of system components.

So, a *compositional* supervisory control problem is defined by a set of system components and a partition of the event set into *controllable* and *uncontrollable* events. The idea is to synthesise a so-called *supervisor* module that disables some of the controllable events, while monitoring the uncontrollable ones, so that every word (i.e., behavior) generated by the *restricted* plant can be extended to a word in the language marked/accepted by the plant (i.e., a "good" behavior).

A word w belongs to the language generated by T restricted by a function $\sigma : A_T^* \mapsto 2^{A_T}$ (denoted $\mathcal{L}^\sigma(T)$) if every prefix of w "survives" σ. More formally, $w = \ell_0, \ldots, \ell_k \in \mathcal{L}^\sigma(T)$ *if and only if* $t_0 \xrightarrow{\ell_0 \ldots \ell_i}_T t_{i+1}$ and $\ell_i \in \sigma(\ell_0, \ldots, \ell_{i-1})$, for all $0 \le i \le k$. As expected, $\mathcal{L}_m^\sigma(E) = \mathcal{L}^\sigma(E) \cap \mathcal{L}_m(E)$.

Definition 3 (Compositional Supervisory Control Problem)**.** A *Compositional Supervisory Control Problem* is a pair $\mathcal{E} = (E, A_C)$, where E is a set of automata $\{E_0, \ldots, E_n\}$ (we will abuse notation and use E to also refer to the composition $E_0\| \ldots \|E_n$), the *plant*, and $A_C \subseteq A_E$ is the set of *controllable events* (i.e., $A_U = A_E \setminus A_C$ is the set of uncontrollable events). A solution for $\mathcal{E}$ is a supervisor $\sigma : A_E^* \mapsto 2^{A_E}$ that is:

- *controllable*, that is, $A_U \subseteq \sigma(w)$ for all $w \in A_E^*$; and
- *non-blocking*, that is, for every word $w \in \mathcal{L}^\sigma(E)$ there exists a non-empty word $w' \in A_E^*$ such that $ww' \in \mathcal{L}_m^\sigma(E)$.

In words, a supervisor σ is said to be controllable (or admissible) if it only disables controllable events; and it is non-blocking if it is able to restrict the language generated by the plant E to a set of strings that can always be extended to reach a marked "good" state. Note that the non-blocking property does not guarantee that a marked state will eventually be reached, as uncontrollable events may prevent so (but they will never lead to a deadlock).

The fact that we explicitly define a control problem as taking a compositional description of the plant is important. First, we do this because some supervisory control techniques (e.g. [**?**], [**?**]) avoid the blowup produced by the parallel composition by reasoning directly on the individual components. Secondly, when taking a compositional approach, it is not longer necessary to require the usual desired *specification language* $\mathcal{K} \subseteq \mathcal{L}_m(E)$ used in monolithic formalizations of SC. Indeed, one can just force the specification by defining an auxiliary "observer" automaton K such that $\mathcal{L}_m(E\|K) = \mathcal{K}$.

Finally we point out that, traditionally, SC techniques look for so-called *maximally permissive* supervisors, that is, supervisors that do not disable more events than needed. Maximality comes at a cost in complexity, and therefore there is a growing interest in less ambitious solution concepts, such as *directors* [**?**], which require supervisor controllers that enable at most one[1] controllable event at each state. In our work, we follow this trend towards non-maximality and accept a supervisor satisfying solely the base requirements. As a matter of fact, even obtaining a non-maximal supervisor may still require an exponential amount of time [**?**].

Example 1. *Figure 1 shows an example of a compositional supervisory control problem where two automata model a manufacturing plant that, upon request, produces one of two products. Automaton C represents a customer that can – uncontrollably – request one of two products (events r_1 or r_2), and then wait for its delivery (d_1 or d_2 respectively). Automaton F represents a factory that can produce the products (p_1 or p_2) and then deliver them (d_1 or d_2, in sync with C). The "goal" is to deliver products; hence, we mark states where products have been delivered (c_0 and f_0).*

Automata $C\|F$ depicts the joint behavior of the customer and the factory. Observe that F is not able to produce a new product until the previous one has been delivered. Thus, the composition $C\|F$ could reach a deadlock situation – preventing the goal – if a product of the wrong type is produced (i.e., states $\langle c_1, f_2\rangle$ and $\langle c_2, f_1\rangle$). The supervisor S – depicted also as an automaton – avoids such deadlocks by disallowing the production of non-requested products; it is controllable and non-blocking.

B. Reactive Synthesis

Reactive synthesis is concerned with the realization of a reactive module from a specification given in a temporal logic, and assuming an adversarial environment. We follow [**?**] and consider the *realizability* of CTL* specifications over a set of Boolean variables V, where $V = In \,\dot\cup\, Out$ is partitioned into a set of *input* signals In and a set of *output* signals Out.

[1] In a "mixed" state with possible controllable and uncontrollable events, the controller may decide to disable all controllable events and just wait for an uncontrollable event to occur.

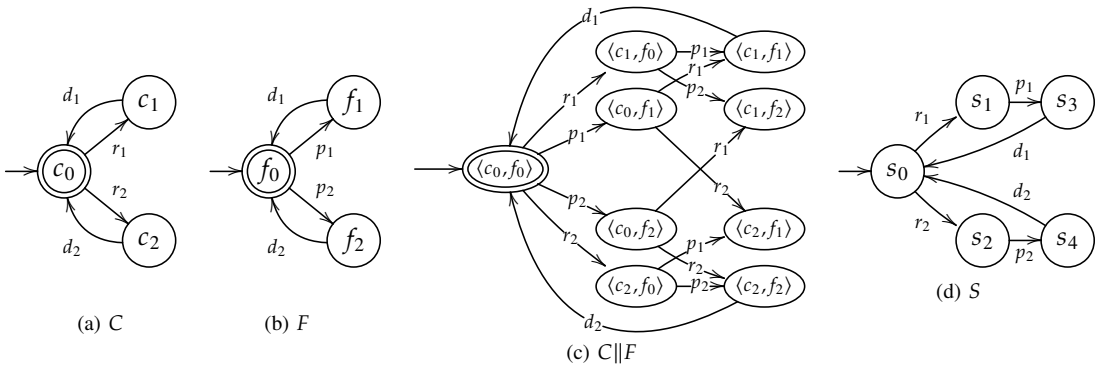


Figure 1. Compositional supervisory control problem example and solution with controllable events $\{d_1, d_2, p_1, p_2\}$.

Definition 4 (CLT* Formula)**.** A *CLT* formula* over Boolean variables V is either a state formula Ψ or a path formula Φ, both defined inductively as follows:

$$\begin{aligned} \Psi &= \mathit{true} \mid v \mid \mathbf{E}\,\Phi \quad (\text{with } v \in V); \\ \Phi &= \Psi \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathbf{X}\,\Phi \mid \Phi\ \mathbf{U}\ \Phi, \end{aligned}$$

The *path quantifier* $\mathbf{E}\,\Phi$ states that there exists exists a path for which Φ holds (possibility), whereas $\mathbf{X}$ and $\mathbf{U}$ in path formulas stand for next and until *temporal operators*, respectively.

As it is usual, we consider *false*, $\wedge$, $\Rightarrow$, $\mathbf{A}$ (inevitably; meaning for all paths), $\mathbf{G}$ (globally or always) and $\mathbf{F}$ (finally or eventually) as convenient abbreviations:

$$\begin{aligned} \mathit{false} &\Leftrightarrow \neg\mathit{true} \\ \Phi \wedge \Psi &\Leftrightarrow \neg(\neg\Phi \vee \neg\Psi) \\ \Phi \Rightarrow \Psi &\Leftrightarrow \neg\Phi \vee \Psi \\ \mathbf{A}\,\Phi &\Leftrightarrow \neg\,\mathbf{E}\,\neg\Phi \\ \mathbf{G}\,\Phi &\Leftrightarrow \neg\,\mathbf{F}\,\neg\Phi \\ \mathbf{F}\,\Phi &\Leftrightarrow \mathit{true}\ \mathbf{U}\ \Phi \end{aligned}$$

The semantics of a CTL* formula is defined with respect to a Kripke structure $K = (S, R, L)$, where S is the set of states, $R \subseteq S \times S$ is a total transition relation, and $L : S \mapsto 2^V$ is a function that labels states with the set of variables true in the state. A path in K from a state s_0 is an infinite sequence of states $p = s_0, s_1, \ldots$, such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.

We consider the following notation:

- $\Pi(K, s)$ denotes the set of all paths in K from state s;
- p_i denotes the i-th element of path p;
- $p|^j$ denotes the suffix of path p starting at s_j;
- $K, s \models \Psi$ denotes that state formula Ψ holds at state s; and
- $K, p \models \Phi$ denotes that path formula Φ holds along path p.

The relation $\models$ is defined inductively as follows:

$$\begin{aligned} &K, s \models \mathit{true} \\ &K, s \models v && \Leftrightarrow\ v \in L(s) \\ &K, s \models \mathbf{E}\,\Phi && \Leftrightarrow\ \exists\, p \in \Pi(K, s)\ .\ K, p \models \Phi \\ &K, p \models \Psi && \Leftrightarrow\ K, p_0 \models \Psi \\ &K, p \models \neg\Phi && \Leftrightarrow\ \neg(K, p \models \Phi) \\ &K, p \models \Phi_1 \vee \Phi_2 && \Leftrightarrow\ (K, p \models \Phi_1) \vee (K, p \models \Phi_2) \\ &K, p \models \mathbf{X}\,\Phi && \Leftrightarrow\ K, p|^1 \models \Phi \\ &K, p \models \Phi_1\ \mathbf{U}\ \Phi_2 && \Leftrightarrow\ \exists\, j\ .\ (K, p|^j \models \Phi_2) \wedge \\ & && \quad\ \forall\, 0 \leq k < j\ .\ (K, p|^k \models \Phi_1) \end{aligned}$$

Generally speaking, realizability for reactive synthesis problems amounts to checking whether there exists an open controller that satisfies a CTL* specification. The problem is best seen as a *two-player game* [**?**] between players $\mathcal{I}$ (the environment) and $\mathcal{II}$ (the system). The game is played in turns; player $\mathcal{I}$ starts by giving a subset of the variables in *In* and player $\mathcal{II}$ responds with a subset of the variables in *Out*. The players play according to *strategies*: a mapping $\delta_{\mathcal{I}} : (2^{Out})^* \mapsto 2^{In}$ for player $\mathcal{I}$, and a mapping $\delta_{\mathcal{II}} : (2^{In})^+ \mapsto 2^{Out}$ for player $\mathcal{II}$. We use δ to denote the pair of strategies $\langle \delta_{\mathcal{I}}, \delta_{\mathcal{II}} \rangle$.

The strategies induce the structure $K^\delta = (S, R, L)$, where:

- $S = (2^V)^*$ is the (infinite) set of states containing all possible finite sequences of the form $(s_0, \ldots, s_n)$, with $n \geq 0$, and $s_i \in 2^V$ for all $0 \leq i \leq n$;
- $((s_0, \ldots, s_n),\ (s'_0, \ldots, s'_m)) \in R$ *if and only if* $m = n + 1$, $s_i = s'_i$, for all $0 \leq i \leq n$, and $s'_m = s^{\mathcal{I}}_m \cup \delta_{\mathcal{II}}(s_0 \backslash Out, \ldots, s_n \backslash Out, s^{\mathcal{I}}_m)$, where $s^{\mathcal{I}}_m = \delta_{\mathcal{I}}(s_0 \backslash In, \ldots, s_n \backslash In)$; and
- the labeling function is $L((s_0, \ldots, s_n)) = s_n$ (i.e., the last state in the sequence).

Definition 5 (CTL* Realizability)**.** A CTL* formula Φ over variables $V = In \dot\cup Out$ is *realizable* from an initial state $s_0 \in 2^V$ if there *exists* a strategy $\delta_{\mathcal{II}} : (2^{In})^+ \mapsto 2^{Out}$ for the system player such that for *every* possible strategy $\delta_{\mathcal{I}} : (2^{Out})^* \mapsto 2^{In}$ of the environment player, it is the case that $K^\delta, s_0 \models \Phi$.

Realizability for general CTL* formulas is known to be a difficult problem (i.e., 2EXPTIME-complete [**?**]). However, it is possible to bypass this complexity by restricting to the CTL subset of formulas, in which temporal operators must be immediately preceded by a path quantifier. In this paper we focus on CTL, since it is sufficiently expressive to capture supervisory control goals, allowing us to leverage on EXPTIME synthesis procedures [**?**].

We note that reactive synthesis assumes a synchronous execution model, in which environment and system execute in a lockstep manner. In contrast, SC uses an asynchronous model, as there is no assumption on the relative execution speeds of the intervening components. Furthermore, while in reactive synthesis multiple signals may change simultaneously at a given clock tick, only one of the enabled events may ensue from a given state under the SC framework.

C. Automated Planning

Automated planning is a model-based approach to the synthesis of plans built from primitive actions (referred to as operators) so as to bring about a given goal in a domain. The dynamics of the domain is specified with a factored representation describing the preconditions and effects of actions available. For our needs, we shall focus here on the Fully Observable Non-Deterministic (FOND) variant [**?**], which extends classical planning with non-deterministic effects.

Definition 6 (FOND Planning)**.** A *FOND planning problem* is a tuple $\mathcal{P} = (F, I, O, G)$, where F is a set of *fluent* propositions, $I \subseteq F$ is a set (conjunction) of fluents encoding the *initial state*; O is a set of *operators*; and $G \subseteq F$ is a set (conjunction) of fluents defining the *goal* to be achieved.

An *operator* is a pair $o = (Pre(o), Eff(o))$, where $Pre(o)$ is a Boolean formula over F describing the *preconditions* of the operator, and $Eff(o)$ is a conjunction of non-deterministic effects. A *non-deterministic effect* has the form $e_1 \mid \ldots \mid e_n$, where each e_i is a conjunction of deterministic conditional effects (when $n = 1$ the effect is said to be deterministic). A *conditional effect* has the form $C \Rightarrow E$, where C is a Boolean formula over F and E is a *literal* over F (i.e., f or $\neg f$).[2]

A *state* is a subset of F (or conjunction of fluents) representing those fluents that are true in the state. Whenever an operator o with a non-deterministic effect $e_1|\cdots|e_n$ is executed, exactly one of the e_i effects will ensue non-deterministically, that is, without the control of the executor. In turn, a conditional effect $C \Rightarrow E$ states that when condition C holds in the state in which the action is being executed, the set of literals E ought to hold in the successor state (and everything else remains the same). With this understanding, it is possible to define a function $Succ(o, s)$ denoting the *possible successor states* when operator o is executed in a state s [**?**].

A solution to a FOND planning task is a *policy* $\pi : 2^F \mapsto 2^O$ that maps a state s to a set of "appropriate" operators $\pi(s)$ such that, under plausible assumptions, the goal is eventually reached [**?**]. Formally, we say that a state s' is *reachable* from a state s by following a policy π if there exists a sequence $s_0, s_1, \ldots, s_k$ such that $s_0 = s$, $s_k = s'$, and $s_{i+1} \in Succ(\pi(s_i), s_i)$, for all $0 \leq i < k$. A policy π is *closed* if $\pi(s) \neq \emptyset$ for every state s reachable from I by following π. Then, a policy π is a solution—a *strong-cyclic* plan—for a planning problem $\mathcal{P} = (F, I, O, G)$ if and only if π is closed, and every reachable state from I by following π can reach a goal state, again, using the policy. A strong cyclic plan guarantees the goal under the fairness assumptions that all possible effects of an action will ensue in a state if executed infinitively often in that state [**?**], [**?**]. While other solution concepts have been proposed for FOND planning,[3] we shall focus on strong cyclic plans since, as we will later see, they better reflect the non-blocking requirement in SC: we look for plans in which the goal is always reachable, but not necessarily in a finite number of steps (i.e., may enter a loop that cannot be escaped only by controllable choices).

We close by noting two major differences between planning and SC. First, planning does not provide an account of uncontrollable events: all actions are produced by the planner/executor. Second, planning is concerned with reaching a desired final state, and behavior is meant to stop then; whereas SC is interested in continuous "good" behavior (of the plant).

[2]This formalization of actions corresponds to the UC Normal Form [**?**] with no nested conditional effects.

[3]Interestingly, other solution concepts are of interested in FOND planning. For example, a *strong plan* solution is a closed policy guaranteeing the goal in a bounded finite number of steps in spite of non-determinism.

IV. Supervisory Control via Reactive Synthesis

In this section we develop a translation from a deterministic supervisory control (Definition 3) to reactive synthesis (Definition 5). In doing so, we need to address the misalignment between their execution models (asynchronous/synchronous) and step-changes allowed (single/multiple), as discussed above.

Consider a compositional supervisory control problem $\mathcal{E} = (\{E_0, \ldots, E_n\}, A_C)$, with components $E_i = (S_{E_i}, A_{E_i}, \rightarrow_{E_i}, e_0^i, M_{E_i})$ and controllable events $A_C \subseteq A_E = \bigcup_{i=0}^{n} A_{E_i}$. Following [**?**], we show how to encode $\mathcal{E}$ as a CTL formula:

$$\varphi_{\mathcal{E}} = (\Psi_0 \wedge \Psi_A) \Rightarrow (\Psi_G \wedge \Psi_W)$$

where:

- Ψ_0 represents the initial condition;
- Ψ_A represents the environment assumptions;
- Ψ_G represents the system guarantees; and
- Ψ_W represents the goal.

Importantly, the translation is such that each transition relation $\rightarrow_{E_i}$ is modeled separately, thus *avoiding the computation of the parallel composition*; and the three rules (see Definition 2) for parallel composition are modeled explicitly as part of the reactive synthesis problem.

For legibility, we use variables over finite domains, under the understanding that they can always encoded using enough Boolean variables. Also, if x is a finite-domain variable, we use $x \in X$ as a shorthand of $\bigvee_{c \in X} (x = c)$.

Variables. The set of input $In = \mathsf{S} \cup \{\mathsf{u}\}$ and output signals $Out = \{\mathsf{c}\}$ are as follows:

- $\mathsf{S} = \{\mathsf{s}_0, \ldots, \mathsf{s}_n\}$, with each signal variable s_i ranging over S_{E_i} and encoding the current state of automaton E_i.
- Signal variable u ranging over $A_U \cup \{\lambda\}$ and encoding the environment's choice of an uncontrollable event or no choice (via special value λ).
- Signal variable c ranging over A_E and encoding the system's choice of any event (controllable or not).

Initial Condition. Initially, all system components are in their corresponding initial states:

$$\Psi_0 = \bigwedge_{i=0}^{n} (\mathsf{s}_i = e_0^i).$$

Goal. We require marked states – of the composed system – to be always reachable, that is, for all paths there always exists a path where a marked state is eventually reached):

$$\Psi_W = \mathbf{A}\,\mathbf{G}\left(\mathbf{E}\,\mathbf{F}\left(\bigwedge_{i=0}^{n} (\mathsf{s}_i \in M_{E_i})\right)\right).$$

Recall, from Definition 2, that for a state of the composition to be marked, the states of all components have to be so.

Macros. We define here two convenient abbreviations to capture two important aspects of a DES. First, the macro *enabled*(ℓ) states whether the shared event ℓ is "ready" for execution, in that ℓ may occur next in every automaton that has such event in its alphabet (i.e. it synchronizes):

$$\mathit{enabled}(\ell) = \bigwedge_{\{E_i \in E \mid \ell \in A_{E_i}\}} \mathsf{s}_i \in \{e \mid e \xrightarrow{\ell}_{E_i} e'\}.$$

Second, abbreviation $\mathit{step}(\ell, E_i)$ models a step of component E_i via event ℓ.

$$\mathit{step}(\ell, E_i) = \begin{cases} \bigwedge_{(e,\ell,e') \in \rightarrow_{E_i}} \big((\mathsf{s}_i = e) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_i = e')\big) & \text{if } \ell \in A_{E_i} \\ \bigwedge_{e \in S_{E_i}} \big((\mathsf{s}_i = e) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_i = e)\big) & \text{otherwise.} \end{cases}$$

Note that only macro *step* uses a temporal operator (i.e., $\mathbf{A}\,\mathbf{X}$), while *enabled* uses only Boolean connectives.

With these abbreviations at hand, we are ready to define the assumption and guarantee formulas.

Guarantees. In the guarantee formula, we encode the requirement that the controller should only select valid controllable events:

$$\Psi_G = \bigwedge_{\ell \in A_E} \mathbf{A}\,\mathbf{G}\big((\mathsf{c} = \ell) \Rightarrow \mathit{enabled}(\ell)\big).$$

That is, the system is required to guarantee that the events it chooses are possible in every component mentioning the event.

Assumptions. In the assumption formula, we encode the transition relation $\rightarrow_E$ such that when an event is selected (be it controllable or uncontrollable), the corresponding step is executed (synchronously in each automaton). Formula Ψ_A is built from the conjunction of the following formulas:

(1) $\mathbf{A}\,\mathbf{G}\big((\mathsf{u} = \ell) \Rightarrow \mathit{enabled}(\ell)\big)$, for each $\ell \in A_U$, asserting that if the environment chooses an uncontrollable event ℓ, then every component mentioning such event is enabled (i.e., in a state in which the event may occur).

(2) $\mathbf{A}\,\mathbf{G}\big((\mathsf{u} = \ell) \Rightarrow \mathit{step}(\ell, E_i)\big)$, for each component $E_i \in E$ and $\ell \in A_U$, asserting that when the environment chooses the uncontrollable event ℓ, then the current state of component E_i (encoded in variable s_i) should be updated to reflect a step over event ℓ.

(3) $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = \ell) \wedge (\mathsf{u} = \lambda) \Rightarrow \mathit{step}(\ell, E_i)\big)$, for each $E_i \in E$ and event $\ell \in A_E$, asserting that when the the environment "skips" (i.e., $\mathsf{u} = \lambda$) and the system chooses event ℓ, then the current state of component E_i should be updated to reflect a step over event ℓ. The particular case in which the system c happens to select a *uncontrollable* event ℓ represents its decision to disable all controllable events and just "wait" for an uncontrollable event to occur. Note it is irrelevant which uncontrollable event is picked, because all possible cases have already been covered by the formulas above and, as a result, a solution strategy still needs to work for any of them.

Observe that, in the worst case (when there are transitions for every pair of states and for each event), the complexity of the translation is $O\big((\sum_{i=0}^{n}|A_{E_i}|)(\sum_{i=0}^{n}|S_{E_i}|^2)\big)$, since:

- in order to generate transitions, assumptions and guarantees, each event needs to be considered *twice* (once to determine the validity of selecting the event, and once to either enforce the corresponding state updates or encode the fact that a state does not change); and
- one variable s_i captures the state E_i is at, and variables (u and c) encode the selection of the event to execute.

Example 2. *Consider the compositional supervisory control problem $\mathcal{E} = (\{C, F\}, A_C)$, where C and F are the automata depicted in Figures 1.a and 1.b, respectively, and $A_C = \{d_1, d_2, p_1, p_2\}$ and $A_U = \{r_1, r_2\}$ are the sets of controllable and uncontrollable events, respectively. We show a fragment of the translation for the such problem. We use the variables s_C and s_F to encode the current states of automata C and F, and variables c and u to encode the selection of controllable and uncontrollable events to execute, if any. For legibility, we restrict to formulas over events d_1, p_1, and r_1 (formulas for events d_2, p_2 and r_2 are analogous).*

Initial Condition: $(\mathsf{s}_C = c_0) \wedge (\mathsf{s}_F = f_0)$

Goal: $\mathbf{A}\,\mathbf{G}\,\big(\mathbf{E}\,\mathbf{F}(\mathsf{s}_C \in M_C \wedge \mathsf{s}_F \in M_F)\big)$

Assumptions:

- $\mathbf{A}\,\mathbf{G}\big((\mathsf{u} = r_1) \Rightarrow \mathit{step}(r_1, C)\big)$, *where* $\mathit{step}(r_1, C) = \big((\mathsf{s}_C = c_0) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_C = c_1)\big)$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{u} = r_1) \Rightarrow \mathit{step}(r_1, F)\big)$, *where* $\mathit{step}(r_1, F)$ *keeps the value of* s_F, *since* $r_1 \notin A_F$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = r_1) \wedge (\mathsf{u} = \lambda) \Rightarrow \mathit{step}(r_1, C)\big)$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = r_1) \wedge (\mathsf{u} = \lambda) \Rightarrow \mathit{step}(r_1, F)\big)$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = d_1) \wedge (\mathsf{u} = \lambda) \Rightarrow \mathit{step}(d_1, C)\big)$, *where* $\mathit{step}(d_1, C) = \big((\mathsf{s}_C = c_1) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_C = c_0)\big)$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = d_1) \wedge (\mathsf{u} = \lambda) \Rightarrow \mathit{step}(d_1, F)\big)$, *where* $\mathit{step}(d_1, F) = \big((\mathsf{s}_F = f_1) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_F = f_0)\big)$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = p_1) \wedge (\mathsf{u} = \lambda) \Rightarrow \mathit{step}(p_1, C)\big)$, *where* $\mathit{step}(p_1, C)$ *keeps the current value of* s_C, *since* $p_1 \notin A_C$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = p_1) \wedge (\mathsf{u} = \lambda) \Rightarrow \mathit{step}(p_1, F)\big)$, *where* $\mathit{step}(p_1, F) = \big((\mathsf{s}_F = f_0) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_F = f_1)\big)$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{u} = r_1) \Rightarrow \mathit{enabled}(r_1)\big)$, *where* $\mathit{enabled}(r_1) = (\mathsf{s}_C = c_0)$.

Guarantees:

- $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = d_1) \Rightarrow \mathit{enabled}(d_1)\big)$, *where* $\mathit{enabled}(d_1) = (\mathsf{s}_C = c_1) \wedge (\mathsf{s}_F = f_1)$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = p_1) \Rightarrow \mathit{enabled}(p_1, F)\big)$, *where* $\mathit{enabled}(p_1, F) = (\mathsf{s}_F = f_0)$.
- $\mathbf{A}\,\mathbf{G}\big((\mathsf{c} = r_1) \Rightarrow \mathit{enabled}(r_1, C)\big)$.

We note that the above scheme of assumptions-guarantees is inspired by the GR(1) [**?**] restrictions for LTL formulas, for which efficient synthesis procedures have been proposed. Under this scheme, if the environment purposely violates an assumption, the formula is trivially satisfied (since assumptions are the antecedent of an implication). Also, reaching a deadlock configuration makes the guarantee of selecting an enabled event unsatisfiable. This is correct since reaching a deadlock prevents fulfilling the non-blocking requirement.

Interestingly, the translation also satisfies other restrictions imposed by GR(1), including limited nesting of temporal operators, and appropriate updates of variables. Nonetheless,

the non-blocking requirement of SC cannot be expressed in GR(1), as the goal formula Ψ_W requires the use of the existential path quantifier **E**, not present in LTL. Remarkably, however, replacing **E** with **A** in such goal formula, we obtain an LTL specification expressing a stronger goal requiring to effectively reach marked states (i.e, $\mathbf{G}(\mathbf{F}(\mathit{marked}))$), which is also a relevant type of requirement studied in SC [**?**]. Thus, the approach of targeting an LTL solver with this altered translation is sound but not complete. That is, a stronger solution obtained by an LTL solver would also be a solution for the SC problem, but not finding a solution would not imply that a "weaker" non-blocking solution does not actually exists.

The following result shows that the encoding is correct.

Theorem 1. Let $\mathcal{E}$ be a compositional supervisory control problem and let $\varphi_{\mathcal{E}}$ the CTL formula obtained by following the above proposed translation from $\mathcal{E}$. Then, there exists a supervisor σ for $\mathcal{E}$ if and only if there exists a strategy δ for $\varphi_{\mathcal{E}}$. In addition, σ can be constructed from δ, and vice versa.

Proof. We prove this by induction on the number of automata. We start by considering the monolithic supervisory control problem $\mathcal{M} = (E_0, A_C)$, that is for a single automaton E_0 (or equivalently a singleton set). In this case the above translation yields a CTL formula $\varphi_{\mathcal{M}}$. Let us assume that there exists a supervisor for $\mathcal{M}$, since $\mathcal{M}$ is a deterministic problem then – as proved in [**?**] – there is director $\sigma_{\mathcal{M}}$ for $\mathcal{M}$ (i.e., a supervisor for $\mathcal{M}$ that enables at most one controllable event for each state). Let $w = \ell_0, \ldots, \ell_k$ be a word in $\mathcal{L}^{\sigma_{\mathcal{M}}}(E_0)$, then there is a sequence of steps over w, restricted by $\sigma_{\mathcal{M}}$ and starting at the initial state e_0^0 of E_0:

$$e_0^0 \xrightarrow{\ell_0}_{E_0} \ldots \xrightarrow{\ell_k}_{E_0} e_{k+1}^0$$

Starting from the initial state e_0^0 we can make the following observations for each step $e_i^0 \xrightarrow{\ell_i}_{E_0} e_{i+1}^0$ in the sequence:

- Initially, variable s_0 encodes state e_i^0 (true for state e_0^0 in the initial condition Ψ_0).
- If $\ell_i \in A_U$ then by rule (1) of Ψ_A and by Ψ_G variables u and c can be set to select ℓ_i since it is *enabled* at e_i^0.
- If $\ell_i \in A_C$ then by rule Ψ_G variable c can be set to encode ℓ_i since it is *enabled* at e_i^0 and by rule (3) it must be the case that $\mathsf{u} = \lambda$ (otherwise a race condition would allow the environment to act instead).
- Finally, variable s_0 is updated to encode state e_{i+1}^0, by rule (2) if $\mathsf{u} = \ell_i$ or by rule (3) if $\mathsf{c} = \ell_i$ and $\mathsf{u} = \lambda$.

Therefore, a state is reachable in E_0 restricted by $\sigma_{\mathcal{M}}$ if and only if it is also reachable in a path of $\varphi_{\mathcal{M}}$. Then, since $\sigma_{\mathcal{M}}$ guarantees that a marked state can be reached from e_k^0 (i.e., non-blocking), the corresponding path satisfies the goal described in $\varphi_{\mathcal{M}}$. Ergo, a strategy $\delta_{\mathcal{M}}$ obtained by extracting the sequence of observed events w from its input, and returning the encoding of $\sigma_{\mathcal{M}}(w)$, is a winning strategy for $\varphi_{\mathcal{M}}$.

We can extract the sequence of observed events by projecting the event encoded by u if it is not λ, or the event encoded by c otherwise. Whereas, encoding $\sigma_{\mathcal{M}}(w)$ can be done by setting $\mathsf{c} = \sigma_{\mathcal{M}}(w)$ when non-empty, or by setting some enabled uncontrollable event otherwise (since none of these events can prevent the goal).

The opposite direction is similar. That is, for any path in the Kripke structure induced by $\varphi_{\mathcal{M}}$ following strategy $\delta_{\mathcal{M}}$, there is a corresponding run in E_0. By rule (1) of Ψ_A and by Ψ_G only enabled events can be selected for execution. By rule (2) transitions through uncontrollable events occur independently of the value of c, while by rule (3) transitions through controllable events occur only when $\mathsf{u} = \lambda$ (i.e., not a race condition). In other words, the strategy cannot disable uncontrollable events, complying with the controllability requirement. Then, along a path satisfying $\varphi_{\mathcal{M}}$ it is always the case that a marked state is reachable, also meeting the non-blocking requirement. Therefore, given a strategy $\delta_{\mathcal{M}}$ for $\varphi_{\mathcal{M}}$, we can build a supervisor $\sigma_{\mathcal{M}}$ by extracting the sequence of uncontrollable events w from $\sigma_{\mathcal{M}}$'s input and returning the event encoded by $\delta_{\mathcal{M}}(w)$ if controllable, or an empty set if uncontrollable.

For the inductive case we assume the property holds for the compositional supervisory control problem $(\{E_0, \ldots, E_{n-1}\}, A_C)$. Hence, the compositional problem is equivalent to the monolithic problem over the automaton $E = E_0 \| \ldots \| E_{n-1}$. We use this to check that the property holds for the original set plus E_n, much like in the base case.

As before, it can be shown that a state is reachable in $E \| E_n$ if and only if it is reachable in a path of $\varphi_{\mathcal{E}}$, and we can build a strategy $\delta_{\mathcal{E}}$ from a director $\sigma_{\mathcal{E}}$. The opposite direction also is analogous. That is, $\varphi_{\mathcal{E}}$ effectively captures the semantics of $\mathcal{E}$, taking the rules of parallel composition into account. Therefore, once again we can build a supervisor $\sigma_{\mathcal{E}}$ from a strategy $\delta_{\mathcal{E}}$. □

V. Supervisory Control via Automated Planning

We now turn to the problem of seeing a compositional supervisory control problem (Definition 3) as a planning task (Definition 6). The translation is close to that to reactive synthesis, but capturing uncontrollable events through *non-determinism* and continuous (never ending) behavior through planning for "recurrent" goals, thus addressing the "mismatches" described at the end of Section III-C. The key ingredients of the encodings are as follows:

1) Each transition relation $\rightarrow_{E_i}$ is modeled separately, thus *avoiding the computation of the parallel composition*.
2) Composition rules are modeled explicitly (within the problems to be solved), by encoding them within the actions preconditions and effects.
3) Each (controllable or uncontrollable) event ℓ_c is mapped to an operator ℓ available to the planner.
4) When an uncontrollable event is select at a so-called "non-deterministic choice phase," the planner will be *forced to take* the corresponding operator.
5) For simplicity, we do not require the non-deterministic choice of uncontrollable events to choose a valid (enabled) event. Thus, when no valid event is selected we default the choice to the planner. This does not grant more control to the planner, as it still ought to consider (and resolve) every possible non-deterministic choice.

6) We request a *strong cyclic* plan reaching a marked state, and then use the approach presented in [**?**] to extend the goal with recurrence.

In what follows, consider the compositional supervisory control problem $\mathcal{E} = (\{E_0, \ldots, E_n\}, A_C)$, with components $E_i = (S_{E_i}, A_{E_i}, \rightarrow_{E_i}, e_0^i, M_{E_i})$, for $0 \leq i \leq n$, and controllable events $A_C \subseteq A_E$ (with $A_U = A_E \setminus A_C$ being the set of uncontrollable events). We first develop an intermediate planning problem $\mathcal{P}_\mathcal{E} = (F, I, O, G)$ encoding $\mathcal{E}$ as follows:

Fluents. The set of fluents in $\mathcal{P}_\mathcal{E}$ is defined as:

$$F = \mathsf{S} \cup \{inprogress(\ell_u) \mid \ell_u \in A_U\} \cup \{picked, event\},$$

where:

- $\mathsf{S} = \{\mathsf{s}_0, \ldots, \mathsf{s}_n\}$ such that each fluent $\mathsf{s}_i \in S_{E_i}$ represents the current state of component E_i;[4]
- *event* is a Boolean fluent stating that the last operator executed represented an actual domain event, and not an auxiliary (support) operator;
- $inprogress(\ell_u)$ is a Boolean fluent stating that uncontrollable event ℓ_u has been selected for execution at the last non-deterministic choice phase; and
- *picked* is a Boolean fluent stating that a non-deterministic choice phase has occurred since the last event execution.

Initial State. The initial state of planning problem $\mathcal{P}_\mathcal{E}$ captures the initial states e_0^i of each component E_i:

$$I = \bigwedge_{i=0}^{n} (\mathsf{s}_i = e_0^i) \wedge event.$$

Goal. The goal specification is to reach a planning state representing the arrival to a marked state in the corresponding control problem after a domain event has occured:

$$G = \bigwedge_{i=0}^{n} (\mathsf{s}_i \in M_{E_i}) \wedge event.$$

Operators. The planning domain will contain one operator per event (controllable or uncontrollable) and one distinguished operator pick; hence, $O = A_E \cup \{\mathsf{pick}\}$.

First, operator pick non-deterministically selects an uncontrollable event from set $A_U = \{\ell_0, \ldots, \ell_k\}$ (macro $enabled(\ell)$ remains the same as in Section IV):

$$\begin{aligned} Pre(\mathsf{pick}) =& \neg picked; \\ Eff(\mathsf{pick}) =& picked \wedge \neg event \wedge \\ & \big(enabled(\ell_0) \Rightarrow inprogress(\ell_0) \mid \\ & \quad \vdots \\ & \; enabled(\ell_k) \Rightarrow inprogress(\ell_k) \mid \\ & \; true \Rightarrow true\big). \end{aligned}$$

The key feature of pick is its non-deterministic effect, representing the choice of one of the enabled uncontrollable events, which is then set to be "in-progress" (i.e., forcing the planner to then execute said event). Alternatively, the choice may simply "skip" the selection (i.e., the last conditional effect) setting no event "in-progress" – even when many may be enabled. Independently of the choice, the pick fluent is "toggled" to enable the next execution "phase" (of the selected event, if any).

Secondly, for each *controllable* event $\ell_c \in A_C$ and each *uncontrollable* event $\ell_u \in A_U$, we add operators ℓ_c and ℓ_u, respectively, representing the execution of the event of concern by updating the components' states. By design, we force the planner to select operator ℓ_u when the corresponding uncontrollable event ℓ_u is the one set to "in-progress" (by a previous pick action). This "prioritizes" the environment's choice. Only when no uncontrollable event has been selected for execution, the planner may decide to execute an enabled controllable event; or alternative, the planner may decide to just "wait" for the environment to act (via an enabled uncontrollable event) – as if disabling all controllable events.

The effects for all these operators $\ell_c \in A_C$ and $\ell_u \in A_C$ are almost identical. To capture the evolution of a component E_i with respect to an event ℓ, we reformulate the macro $step(\ell, E_i)$ within the planning framework, relying now on conditional effects (instead of logical implication) and fluent update (instead of the *next* temporal operator). Recall that, in planning, fluents that are not explicitly updated remain unchanged:

$$step(\ell, E_i) = \bigwedge_{(e,\ell,e') \in \rightarrow_{E_i}} \big((\mathsf{s}_i = e) \Rightarrow (\mathsf{s}_i = e')\big).$$

So, for each controllable event $\ell_c \in A_C$ we have a corresponding operator ℓ_c defined follows:

$$\begin{aligned} Pre(\ell_c) &= \bigwedge_{\ell \in A_U} \neg inprogress(\ell) \wedge enabled(\ell_c) \wedge picked; \\ Eff(\ell_c) &= \bigwedge_{\{E_i \in E \mid \ell_c \in A_{E_i}\}} step(\ell_c, E_i) \wedge event \wedge \neg picked. \end{aligned}$$

Similarly, for each uncontrollable event $\ell_u \in A_U$:

$$\begin{aligned} Pre(\ell_u) =& \Big(inprogress(\ell_u) \vee \bigwedge_{\ell \in A_U} \neg inprogress(\ell)\Big) \wedge \\ & enabled(\ell_u) \wedge picked; \\ Eff(\ell_u) =& \bigwedge_{\{E_i \in E \mid \ell_u \in A_{E_i}\}} step(\ell_u, E_i) \wedge event \wedge \neg picked \wedge \\ & \bigwedge_{\ell \in A_U} \neg inprogress(\ell). \end{aligned}$$

That is, upon the execution of an operator representing a step through a particular event, all relevant components are updated and predicates *picked* and *inprogress* are reset – enabling operator pick again. Also, fluent *event* is made true to signal that a domain event has just occurred. Observe how the design of operators' preconditions implement the priority of uncontrollable events, when one has been chosen by operator pick.

The above produces an intermediate planning problem $\mathcal{P}_\mathcal{E} = (F, I, O, G)$. Notice that the problem produced still does not capture the full semantics of an SC task, as the goal G is simple reachability specification (i.e., to bring about a marked state). In SC, though, the goal is a recurrent one: *always* be able to reach a marked state (even when one has already been reached). To represent such recurrent goal specification,

[4]For simplicity, we treat these as functional fluents over finite domains, which can be automatically encoded via Boolean fluents.

we extend $\mathcal{P}_{\mathcal{E}}$ by applying the technique developed in [**?**] to obtain an extended planning problem $\mathcal{P}'_{\mathcal{E}} = (F', I', O', G')$ that shall implicitly require the construction of a "lasso-loop" after reaching a marked state. Roughly speaking, a new operator loop will "dynamically" designate a marked state – of the composition – as the ulterior goal by "remembering" the designated state via n additional fluents (one per component).

More concretely:

- $F' = F \cup \{\mathsf{s}'_0, \ldots, \mathsf{s}'_n\}$, includes new fluents $\mathsf{s}'_i \in S \cup \{ß\}$ that are used to "save" the selected state to be reached *again* in order to form a lasso (distinguished value ß means no state has yet been selected);
- $I' = I \wedge \bigwedge_{i=0}^{n} (\mathsf{s}'_i = ß)$ extends the original initial situation to the new fluents s'_i, all set to special value ß;
- $O' = O \cup \{\mathsf{loop}\}$ extends the original set of operators with the loop operator, defined as follows:

$$Pre(\mathsf{loop}) = G \wedge \bigwedge_{i=0}^{n} (\mathsf{s}'_i = ß);$$

$$Eff(\mathsf{loop}) = \bigwedge_{i=0}^{n} \Big(\bigwedge_{e \in M_{E_i}} \big((\mathsf{s}_i = e) \Rightarrow (\mathsf{s}'_i = e)\big) \Big) \wedge \neg event.$$

  That is, operator loop is only possible in a marked (goal) state and only once (i.e., variables s'_i have not been yet set); and its effect "stores" the current state as per variables s_i into the (additional) auxiliary variables s'_i; and
- $G' = \bigwedge_{i=0}^{n} (\mathsf{s}_i = \mathsf{s}'_i) \wedge event$ is the new goal requiring to reach (again) a previously selected – by operator loop – marked state, thus capturing the infinite recurrence.

Now, a solution to the recurrent planning problem $\mathcal{P}'_{\mathcal{E}}$ will be a strong cyclic policy π of the form $(\pi_1, \mathsf{loop}, \pi_2)$, where π_1 is a strong cyclic policy that can reach a marked state s from the initial state, loop is the auxiliary operator that dynamically designates s as a (recurrent) goal, and π_2 is a strong cyclic policy that, starting from s, and can get to back s. Note that because *event* fluent is false after the execution of loop, the planner is forced to execute at least one more domain event to bring about *event* again. This avoids trivially reaching the chosen recurrent state goal without taking any action.

It is not difficult to see that, in the worst case (i.e., considering transitions connecting every state through every event), the complexity of the above translation is $O\big((\sum_{i=0}^{n}|A_{E_i}|)(\sum_{i=0}^{n}|S_{E_i}|^2)\big)$, since:

- in order to generate the operators each event needs to be considered once; and
- fluents are the union of states in S_{E_i}, plus extra fluents added to "store" a reached marked state, labels of A_U for the *inprogress* fluents, and the *picked* fluent.

Example 3. *Listing 1 shows a fragment of our translation in PDDL for the compositional supervisory control problem $\mathcal{E} = (\{C, F\}, A_C)$, where C and F are the automata depicted in Figures 1.a and 1.b, resp.; and $A_C = \{d_1, d_2, p_1, p_2\}$ and $A_U = \{r_1, r_2\}$ are the sets of controllable and uncontrollable events, resp. Mind that, in PDDL, predicates and logic operators are applied in prefix notation following a LISP-like tradition.*

```
(:init (and (sC=c0) (sF=f0) (event) (sC'=ß) (sF'=ß)))

(:goal (and (sC=sC') (sF=sF') (event)))

(:action pick
  :precondition (not picked)
  :effect (and (picked) (not event)
               (oneof (when (enabled r1) (inprogress r1))
                      (when (enabled r2) (inprogress r2))
                      (when (true) (true)) )))
(:action loop
  :precondition (and (sC = 0) (sF = 0)
                     (sC' = ß) (sF' = ß)
                     (event) )
  :effect (and (when (sC = 0) (sC' = 0))
               (when (sF = 0) (sF' = 0))
               (not event) ))
(:action d1
  :precondition (and (not (inprogress r1))
                     (not (inprogress r2))
                     (enabled d1) (picked) )
  :effect (and (when (sC = c1) (sC = c0))
               (when (sF = f1) (sF = f0))
               (event) (not (picked)) ))
(:action p1
  :precondition (and (not (inprogress r1))
                     (not (inprogress r2))
                     (enabled p1) (picked) )
  :effect (and (when (sF = f0) (sF = f1))
               (event) (not (picked)) ))
(:action r1
  :precondition (and (or (inprogress r1)
                         (and (not (inprogress r1))
                              (not (inprogress r2)) )
                     (enabled r1) (picked) )
  :effect (and (when (sC = c0) (sC = c1))
               (event) (not (picked))
               (not (inprogress r1))
               (not (inprogress r2)) ))
```

Listing 1. Planning translation example.

As one can see, we use the fluents s_C and s_F to encode the states automata C and F are at, and fluents picked and inprogress(ℓ) (for each $\ell \in A_U$) to encode the non-deterministic choice of an uncontrollable event. For presentation purposes we focus on operators for events d_1, p_1 and r_1 (operators for d_2, p_2 and r_2 are analogous); and while we do not expand macros enabled, we do expand macro step.

Observe that operator r_1 only updates the fluent s_C, while operator p_1 only updates the fluent s_F. On the contrary, operator d_1 updates s_C and s_F synchronously. Furthermore, r_1 is uncontrollable (i.e., must be in-progress to be eligible, or no other event must be in-progress), while d_1, and p_1 are controllable (i.e., to be eligible no uncontrollable event can be in-progress).

The example also shows auxiliary operators pick, and loop, which encode the non-deterministic choice of an uncontrollable event (relying on the `oneof` keyword for non-deterministic choices) and recurrence requirements.

While not expanded in the PDDL shown for legibility, the macro enabled boils down to Boolean formulas over fluents s_C and s_F: `(enabled d1)` expands to `(and (sC = c1) (sF = f1))`; `(enabled p1)` expands to `(sF = f0)`; and `(enabled r1)` expands to `(sC = c0)`.

As the result below states, solutions policies for this planning problem are functions corresponding one-to-one with those solving the original SC task, like the one depicted as an automaton in Figure 1d.

The following result states the correctness of the encoding, in that it fully captures the problem of interest.

Theorem 2. Let $\mathcal{E}$ be a compositional supervisory control problem and $\mathcal{P}'_{\mathcal{E}}$ its corresponding planning problem as per the above encoding. Then, there exists a supervisor σ for $\mathcal{E}$ if and only if there exists a strong cyclic policy π for $\mathcal{P}'_{\mathcal{E}}$. In addition, σ can be constructed from π and vice versa.

Proof. As expected, the proof resembles that of Theorem 1. Thus, we focus on the main differences, namely the validity of the translation for the monolithic case, and the mapping between planning policies and supervisors.

We start by considering the monolithic control problem $\mathcal{M} = (E, A_C)$, that is for a single automaton E_0. In this case the above translation yields a planning problem $\mathcal{P} = (F, I, O, G)$. Let σ be a supervisor for $\mathcal{M}$, which we additionally require to be memoryless, that is, σ has to base its decisions solely on the states of the plant. In other words, for any two words w and w' reaching the same state e it holds that $\sigma(w) = \sigma(w')$, and hence we may abuse notation and consider $\sigma(w) = \sigma(e)$. This is without loss of generality since if there is a solution for a deterministic supervisory control problem there exists a memoryless supervisor [**?**].

Let $w = \ell_0, \ldots, \ell_k$ be a word of $\mathcal{L}^\sigma(E_0)$, that is, there is a sequence of steps in E_0 starting at the initial state e_0^0 and restricted by σ:

$$e_0^0 \xrightarrow{\ell_0}_{E_0} \ldots \xrightarrow{\ell_k}_{E_0} e_k^0$$

For every step $e_i^0 \xrightarrow{\ell_i}_{E_0} e_{i+1}^0$ in the sequence we can make the following observations:

- In situation s_i fluent s encodes state e_i^0, fluent *picked* is off, and fluent *event* is on (true for e_0^0 in the initial situation $s_0 = I$).
- Since fluent *picked* is off the only eligible action in s_i is pick, which leads to a situation s'_i where *picked* is on, and *inprogress*(ℓ_i) might be on too since ℓ_i is enabled in e_i^0, or alternatively all *inprogress* fluents might be off.
- If in s'_i fluent *inprogress*(ℓ_i) is on, then ℓ_i is the only eligible action, leading to situation s_{i+1}; in s_{i+1} fluent s encodes e_{i+1}^0, *picked* is off and *event* is on (i.e., reestablishing the original invariant).
- If in s'_i fluent *inprogress*(ℓ_i) is off (all *inprogress* fluents are off), then ℓ_i can be selected by the planner since it is enabled from e_i^0, leading again to situation s_{i+1}.

Therefore, a state e_i^0 is reachable in E_0 restricted by σ if and only if e_i^0 is also reachable in $\mathcal{P}$. Supervisor σ guarantees that a marked state e_m^0 can be reached from e_k^0, consequently in $\mathcal{P}$ a situation s_m representing e_m^0 can be reached from a situation s_k representing e_k^0. Furthermore, since E_0 is finite we can check whether every word in $\mathcal{L}^\sigma(E_0)$ reaching marked state e_m^0 can be extended to reach e_m^0 again; let $E_M \subseteq M_{E_0}$ be the set of such marked states (i.e., E_M represents the states from where it is safe to execute loop). Given the fact that S_{E_0} is finite, E_M cannot be empty since that would make σ violate the non-blocking property.

A strong cyclic policy π for $\mathcal{P}$ can be derived from σ by mapping states to situations (since σ is memoryless), considering with special care auxiliary operators pick and loop. That is, operator pick must be interleaved between operators representing events; and operator loop must be executed when first visiting a state of E_M (i.e., marked states that can be revisited controllably). When operator pick selects an uncontrollable event ℓ_u (i.e., setting it *inprogress*) then π must return ℓ_u next. Otherwise, for a situation s encoding a state $e \in S_{E_0}$, $\pi(s)$ can return $\sigma(e)$. Since σ is non-blocking, every word accepted by $\mathcal{L}^\sigma$ can be extended to reach a marked state, and hence this also holds for π, making it strong cyclic.

The opposite direction is similar. Note that in a strong cyclic policy π for $\mathcal{P}$ operator pick can always choose an enabled uncontrollable event which the planner is forced to take, thus complying with the controllability requirement. This non-deterministic choice may fail to select a valid event, still the policy can enforce the standard progress assumption selecting an enabled uncontrollable event for execution (i.e., forcing the plant to act as by disabling all controllable events). Furthermore, operator loop flags a marked state known to be always reachable, consequently also meeting the non-blocking requirement. Therefore, a memoryless supervisor σ can be derived from π, that is, given a state $e \in S_{E_0}$ and a situation s encoding e we can define $\sigma(e)$ by carefully following $\pi(s)$ (i.e., discarding auxiliary operator loop, plus returning the union of results after every possible outcome of operator pick).

The generalization to the compositional case can be proved inductively on the number of automata in the compositional SC problem. The main consideration to take into account is that the *enabled* and *step* macros need to effectively capture the rules of parallel composition in Definition 2. This is straightforward since *enabled* captures the antecedent of rules in Definition 2 while *step* captures their consequent. □

VI. Evaluation

So far, we have shown how an SC problem can be seen as either a reactive synthesis *or* an non-deterministic planning problem, thus allowing the application of practical techniques and implementations. In this section we report on an evaluation of the proposed translations. The aim of the evaluation is twofold: *(a)* validate our hypothesis that the translations are indeed computationally efficient; and *(b)* test whether our approach, despite its added complexity, successfully allows leveraging tools for reactive synthesis and planning to solve supervisory control problems. To that end, we compare tools native to supervisory control with tools from the other disciplines working under our encodings.

A. Benchmark

We present a benchmark of six problems, three classical to supervisory control (used in the 9th International Workshop on Discrete Event Systems), and three inspired by existing literature. We focus on scalable case studies, since we want to test the applicability of the different techniques with respect to the state explosion problem. In particular, we choose problems that scale up in two different directions, namely the number of intervening components n and the number of states per component k. We do this in order to test how the tools behave – working under our encodings – not only as the number of automata grows, but also as the complexity of the intervening

components increases. Furthermore, we include cases that are not realizable for some combination of parameters, since this is reported to be a worst case scenario for some techniques.

In the following we give a brief description of each problem in the benchmark.[5] For each problem we vary parameters n and k independently between 1 and 6. Hence, the evaluation considers the execution of 36 tests per case study, totaling 216 tests per tool. As a result, the number of reachable states in the composed plants varies from the order of 10^1 states, up to the order of 10^{10} states (depending also on the topology of each problem).

TL (Transfer Line): This is one of the most traditional examples in supervisory control. It consists of n machines $M(1), \ldots, M(n)$ sequentially connected by n buffers $B(1), \ldots, B(n)$ with finite capacity k, and an ending machine TU called Test Unit which may accept or reject a finished product. The goal of the system is to process elements avoiding buffer overflows and underflows. The system controls when to input an element from buffer $B(i-1)$ into machine $M(i)$ (assuming infinite supply from the outside world for $i = 0$); whereas the output from machine $M(i)$ into buffer $B(i)$ is uncontrollable.

DP (Dinning Philosophers): This is the classic concurrency problem where n philosophers sit around a table sharing one fork with each adjacent philosopher. The goal of the system is to control the access to the forks allowing each philosopher to alternate between eating and thinking (i.e., avoiding deadlock and starvation). Additionally, after grabbing a fork a philosopher needs to take k intermediate etiquette steps before eating. Note that the problem has no solution for $n = 1$, since it considers only one fork and a philosopher needs two to eat.

CM (Cat and Mouse): This is a simple game in which n cats and n mice are placed in opposite ends of a corridor divided in $2k + 1$ cells. They take turns to move to one adjacent area at a time, with mice moving first. The goal of the system is to control the movement of mice in order to avoid sharing an area with a cat (the movements of cats are uncontrollable). In the center of the corridor there is a mouse hole, allowing the mice to share the area with the cats safely. Thus, a winning strategy for the mice requires all of them to be always closer to the mouse hole than any cat, in order to be able to evade them.

BW (Bidding Workflow): A company evaluates projects to decide whether to bid for them or not. To do this, a document describing a project needs to be accepted by n teams with different specializations. We want a workflow that attempts to reach consensus, that is, approving the document when all teams accept it, and discarding it when all teams reject it. The document can be reassigned to a team that has rejected it up to k times for revision, but it must not be reassigned to a team that has already accepted it. When a team rejects the document k times it can be discarded even with no consensus.

[5]Specification files available at:
https://github.com/dciolek/compositional-supervisory-control-benchmark

AT (Air-Traffic management): An airport control tower receives up to n simultaneous requests for landing. The tower needs to signal planes whether it is safe to approach the landing ramp or at which of k air-spaces they must perform holding maneuvers. If two planes enter the same ramp or air-space, a crash may occur. The goal is to control the air-traffic to guarantee that all the planes land safely. Notably, the problem only admits solutions when $k > n$: if there are more landing requests than air-spaces where to hold planes, there is no certain way to avoid a crash.

TA (Travel Agency): A travel agency receives requests for vacation packages, and to fulfill them, it interacts with n web-services for the provision of individual amenities (e.g. car rental, flight purchase, hotel booking, etc.). The goal of the system is to orchestrate the web-services in order to provide a complete vacation package when possible, while avoiding to pay for incomplete packages. The protocols for the services may vary (uncontrollably). One variant is the selection of up to k attributes (e.g. flight destination, dates, and class). Another variant in service protocols involves some services requiring a reservation step guaranteeing a purchase order for a short period, while others do not, and hence the purchase may fail.

B. Software tools

In our evaluation we consider the following publicly available software tools, since they are archetypical representatives of each discipline:

From Supervisory Control:

1) MTSA [**?**], implementing monolithic explicit state representation.
2) SUPREMICA [**?**], implementing compositional explicit state representation.

From Reactive Synthesis:

3) MBP [**?**], implementing monolithic symbolic state representation with BDDs.
4) SLUGS [**?**], also implementing monolithic symbolic state representation with BDDs, but optimized for GR(1) goals.

From Planning:

5) MYND [**?**], implementing informed forward search over a explicit state representation.
6) PRP [**?**], implementing offline replanning with generalization of subplans over an explicit state representation.

C. Threats to validity

On the one hand, we point out that comparing tools from different disciplines is a challenging task and, spite our best effort, the comparison might not be completely fair. In particular, traditional supervisory control techniques aim at generating maximally permissive supervisors, and despite the fact that the selected tools (i.e., MTSA and SUPREMICA) *do not guarantee maximality* (SUPREMICA offers to do so *optionally*), they may naturally incur in higher computational costs since, in general, they consider multiple controllable options (i.e., they often find maximal solutions regardless of the selected options, specially for small case studies).

On the other hand, we acknowledge that the results of the automatic translations are structured differently and are usually larger than manually written specifications (despite only growing polynomially). There is a risk that this may hinder the tools working under our encoding. In particular, the planners chosen (i.e., MYND and PRP) rely on extracting useful heuristic information from the input knowledge representation.

When it comes to reactive synthesis, our evaluation includes MBP and SLUGS. MBP is a reactive synthesis tool based on the well-known NuSMV model checker [**?**].[6] It accepts CTL specifications, thus fitting our translation perfectly. In turn, SLUGS is an efficient state-of-the-art reactive synthesis solver for GR(1) goals, a very popular and well-behaved type of LTL specification. Being LTL goals, though, our translation is sound but *not complete* since, as explained before, one is only able to express a goal stronger than the non-blocking requirement (i.e., marked states are guaranteed to be reached infinitely often). Thus, SLUGS may report that no solution exists, even though one does exist. It turns out, however, that such false negatives never occurred in the selected benchmark. Finally, the MTSA tool also implements an optimized algorithm for this particular case of LTL formula, thus bound to the same (theoretical) limitations with respect to completeness.

Taking the above into consideration and despite our tuning attempts to make the most out of every tool, this evaluation cannot be used to establish a categorical superiority of one approach over the other. Yet, it suffices to validate our hypothesis on the applicability of compositional translations.

D. Results

In Table I, we show the time required to perform the translations for each case study (i.e., considering all values for n and k for each case). The experiments were run on a desktop computer with an Intel i7-3770, with 8GB of RAM, and a time-out of 30 minutes. The first important observation is that the time required for the translations is negligible compared to the time required to solve the corresponding problems (translation time takes seconds, while solving time may go beyond timeout limit, sometimes needing hours or days).

	SMV	SLUGS	PDDL
AT	18.3	18.8	18.7
BW	16.4	16.6	16.7
CM	28.1	28.4	26.8
DP	16.2	16.5	16.6
TA	19.2	19.8	19.6
TL	15.7	15.9	16.1

Table I
TRANSLATION TIME PER CASE STUDY AND INPUT FORMAT IN SECONDS.

In Figure 2.a we show results detailed by tool and case study. Each square plots for a particular tool and case study, showing results for the combination of parameters n (number of components) and k (size of components). Black represents a time-out or out of memory. Interestingly, the memory bound is only an issue for MTSA, which may run out of memory when building an explicit representation of the composition.

In Figure 2.b we summarize the results reporting on the totals of solved cases (i.e., no time-outs, out of memory or other failures); and in Figure 2.c we report on the total execution time in minutes (sum of times of every test up to – and including – time-outs).

From these results we make the following observations:

- Planning tools (MYND and PRP), having as input the PDDL files obtained through our translation, performed very well. This suggests that domain-independent heuristic search may be a good alternative to solve compositional supervisory control problems.
- Reactive synthesis tools (MBP and SLUGS), also taking as input the the results of our translation, solved a similar number of instances to supervisory control tools but required more time.
- Despite relying on a compositional analysis, SUPREMICA performs only slightly better than the monolithic approach (implemented by MTSA). Interestingly, in spite of being native to supervisory control, it solves less instances of the benchmark than other tools working under our encoding.
- Around 55% of instances are solved in less than 10 minutes, only 15% are solved between 10 and 30 minutes, while the remaining 30% reaches the time out. This highlights the steep growth in complexity observed as the state space explodes.
- The CM case study proved to be particularly challenging for all tools. Closer inspection revealed that not only the problem space incurs in blowup, but also the solution grows exponentially.
- Scaling up the number of processes n tends to present a bigger challenge than increasing k. This is to be expected since the state space grows as $O(k^n)$.

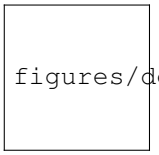
VII. CONCLUSIONS AND FUTURE WORK

In this paper, we explored how to solve compositional supervisory control problems by resorting to automated planning and reactive synthesis techniques. To that end, we developed two corresponding, and related, automatic translations. The key in our translations is the encoding of the compositional aspects of control specifications, resulting in polynomial translations with respect to the size of a compact (compositional) input specification.

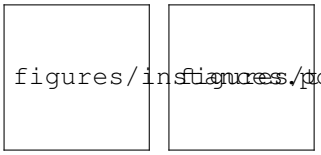
We have found that the best performing tools for the selected benchmark are not tools native to supervisory control, despite the size and artificial structure of the automatically-generated inputs. This indicates that the overhead introduced by our translations is negligible with respect of the cost of solving the actual synthesis task. Nonetheless, the state explosion problem still creates a steep complexity jump that takes tools out of their zone of applicability abruptly.

Naturally, translating supervisory control to reactive synthesis and planning raises the question of whether it is possible (and productive) to perform the inverse translation. Indeed, there are a number of works relying in automata-based intermediate representations for reactive synthesis and planning.

[6] MBP not only supports CTL specifications but also accepts PDDL inputs, which are translated to NuSMV format. However, such translation incurs in an exponential blowup when faced with complex conditional PDDL constructs such as those use by our encoding, turning the approach not practical.



(a) Detailed benchmark results (n is the number of components; k is the size of components).



(b) Number of solved instances.

(c) Total execution time.

Figure 2. Complete benchmark results for the six tools used on the six domains.

However, in general, these are monolithic, and hence subject to a state blowup that is not easy to circumvent.

Likewise, our translations for the deterministic case raise the question of whether it is possible to perform similar translations in the non-deterministic setting. In such setting, there is uncertainty about the state reached after a (non-deterministic) transition, and hence a translation would need to hide the variables used to track states. For this reason the problem would not fit in the FOND planning framework [**?**], and Partially Observable Non-deterministic Planning (PONDP) – a super set of FOND that allows modeling observability – would be required instead. Furthermore, differences between the disciplines' interaction models [**?**] would also be needed to be taken into account.

Beyond the particular tools and benchmark used here, we believe that the potential for cross-fertilization across the three related problems (arising in different communities) is abundant and worth promoting.

Daniel Ciolek is a Ph.D. Student in Computer Science at Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Departamento de Computación and he has a CONICET Ph.D. scholarship working at Instituto de Investigación en Ciencias de la Computación (ICC). His research is focused towards the application of Artificial Intelligence techniques to control problems.

Victor Braberman is an associate professor at Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Departamento de Computación and he is a CONICET researcher working at Instituto de Investigación en Ciencias de la Computación (ICC). Dr. Braberman received a Ph.D. in Computer Science at Universidad de Buenos Aires. His research interests include innovative uses of formal abstractions for the analysis, construction and understanding of software intensive systems. His publication track record includes publications at the top Software Engineering conferences and journals. He serves regularly as PC/PB member for flagship Software Engineering and Formal Methods conferences. He is a current member of IEEE's Transactions on Software Engineering's Editorial Board. He has also significant industrial experience in consultancy and in leading R&D projects for software companies.

Nicolás D'Ippolito is a professor at Universidad de Buenos Aires and a CONICET researcher. He received his undergraduate Computer Science degree from University of Buenos Aires and his Ph.D. in Computing from Imperial College London. His research interests fall in multiple areas Control Theory, Software Engineering, Adaptive Systems and Robotics. Specifically, he is interested in behavior modeling, analysis and synthesis applied to requirements engineering, adaptive and autonomous systems, software architectures design, validation and verification. Dr. D'Ippolito regularly publishes in top conferences and journals in many areas. He has served as PC member for flagship conferences a number of times and has organised many events in top venues.

Sebastián Uchitel received his undergraduate Computer Science degree from University of Buenos Aires and his Ph.D. in Computing from Imperial College London. His research interests are in behavior modeling, analysis and synthesis applied to Software Engineering. Dr. Uchitel was associate editor of the IEEE Transactions on Software Engineering and is currently associate editor of the Springer Requirements Engineering Journal and the Elsevier Science of Computer Programming Journal. He was program co-chair of ASE06 and ICSE10, and General Chair of ICSE17. Dr Uchitel has been distinguished with the Philip Leverhulme Prize, the Konex Foundation Prize and the Houssay Prize.

Sebastián Sardiña is an Associate Professor at RMIT University (Melbourne, Australia) and member of the RMIT Intelligent Agents Group. He received his Ph.D. in the Cognitive Robotics Group at the University of Toronto (Toronto, Canada) on topics related to the Golog family of agent programming languages and reasoning about action and change. Before that, he got his Bachelor in Computer Science at Universidad Nacional del Sur in Bahia Blanca, Argentina. His research in Artificial Intelligence focuses on representation models and reasoning techniques for complex decision making in dynamic systems.