Assume-Guarantee Synthesis of Decentralised Supervisory Control ⋆

Technical Report MPI-SWS-2022-001, April 2022

Ana Maria Mainhardt * **Anne-Kathrin Schmuck** *

* *Max Planck Institute for Software Systems, Kaiserslautern, Germany (e-mail: {amainhardt,akschmuck}@mpi-sws.org).*

Abstract: We address the problem of synthesising *local* supervisors for *decentralised* discrete event systems by designing a framework for the negotiation of assume-guarantee contracts. We set the theoretical foundations of *contract-based supervisory control* and establish the sufficient local conditions – as well as a method to verify and, if needed, enforce them – in order to guarantee nonconflict and cooperation among the synthesised local supervisors, which are also globally maximally permissive if those conditions are immediately satisfied. Our approach has a strong emphasis on *information integrity*: both during synthesis and execution, the dynamics of a subsystem are only partially observed by others through the events they actually share, without the communication of exclusively local events or the use of a coordinator for their supervisors. We have implemented our framework in `Supremica` and show preliminary experimental results.

Keywords: Supervisory Control, Automata, Decentralised Control, Contract Negotiation.

1. INTRODUCTION

Assume-Guarantee (A/G) contracts are a well established paradigm for dealing with large-scale decentralised systems and have a longstanding history and well established theoretical foundations (see e.g. Sangiovanni-Vincentelli et al. (2012); Benveniste et al. (2018)). Its main idea is to decouple dependent decentralised processes, or subsystems, by making use of a set of *compatible local contracts* consisting of an *assumption* and a *guarantee*. If the rest of the system fulfils a process' assumption, this process must ensure its local guarantee holds, which in turn implies that the corresponding assumptions induced on the others also hold. The implications of this methodology are very appealing from a practical perspective, as they allow for (i) efficient design, i.e., local controllers can be synthesised in a decentralised and concurrent fashion; (ii) information integrity, meaning that, apart from what is specified by the contracts, no detailed information about a local behaviour is shared with the rest of the plant; and (iii) decoupled maintenance, as contract-compatible adaptations in a component do not affect others.

Motivated by these desirable features, we develop in this work an assume-guarantee synthesis framework for supervisory control of decentralised discrete event systems. A crucial challenge of employing this paradigm in such setting is the synchronisation of physical components during operation. It is essential, then, not only to design compatible contracts that encode the interaction between the subsystems solely in terms of their shared events, but also to identify and be able to enforce the local conditions that are sufficient to guarantee cooperative control and nonblocking global behaviour.

The method we propose here is inspired by the recent work of Majumdar et al. (2020) in the context of reactive synthesis, and is based on the *negotiation* of contracts. The idea is the following. An initial local supervisor is designed for each process by assuming some cooperation from the rest of the plant, and then translated into contracts to be exchanged between the subsystems. Iteratively, our algorithm refines the local supervisors and generates new contracts to be negotiated. Termination occurs when compatible contracts are achieved and sufficient properties for nonconflict are locally guaranteed. Our framework results, then, in purely local supervisors correctly enforcing a joint global behaviour without a coordinator or communication of exclusively local events.

1.1 Motivating Example

To illustrate our negotiation framework, consider the two interconnected manufacturing lines depicted in Fig. 1. Each line is a chain of different modules (dashed boxes), which in turn consist of linked components such as machines, assembly units and robot manipulators. In addition, multiple buffers (depicted as arrows in Fig. 1) connect neighbouring modules, both within the same line as well as among different lines. Assume that a buffer $\mathtt{B}^{ij}$ may be fed with different kinds of workpieces by $\mathtt{Module}^i$, while $\mathtt{Module}^j$ takes the pieces from this buffer in a first-in-first-out manner; the latter can only control whether to take a piece or not, but it is the former that controls the kind of piece that appears there.

The adjacent modules (or lines) interact only through the synchronisation of shared events, namely, the push/pop events of the buffer(s) connecting them. Nevertheless,

⋆ Both authors are funded through the DFG Emmy Noether grant SCHM 3541/1-1. A. Schmuck is also partially funded through the DFG collaborative research center 389792660 TRR 248 – CPEC.

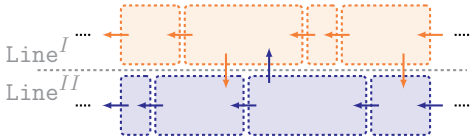

Fig. 1. Illustration of the motivating example.

the different modules do not agree on the controllability status of these events: while $\texttt{Module}^i$ can control the order of workpieces in $\texttt{B}^{ij}$, $\texttt{Module}^j$ cannot. Moreover, the dynamics of $\texttt{Module}^i$ might influence which workpiece is produced and subsequently fed to $\texttt{B}^{ij}$, while the dynamics of $\texttt{Module}^j$ might depend on the order of workpieces got from $\texttt{B}^{ij}$. Hence, given a supervisor for $\texttt{Module}^i$, we can extract a language of workpiece orderings possible in $\texttt{B}^{ij}$, which is a *guarantee* $\texttt{Module}^i$ gives to $\texttt{Module}^j$. The latter, on the other hand, might require a different language of workpiece orderings in $\texttt{B}^{ij}$ to achieve its local specification; this would then be an *assumption* on $\texttt{Module}^i$ induced by $\texttt{Module}^j$. Our negotiation framework iteratively restricts the local supervisors until all assumptions and guarantees over shared events coincide, that is, until each module guarantees what the others assume about its behaviour with respect to the shared buffer, i.e., to the shared events. As we see in Fig. 1, there can be circular dependencies which require multiple rounds of negotiation.

1.2 Related Work and Contribution

Due to the decentralised nature of the resulting local supervisors, our framework has to cope with similar challenges as other decentralised, distributed, or modular supervisor synthesis techniques. One of the main challenges is to ensure *nonconflict*, an intrinsically global property stating that the entire plant in closed loop with its decentralised controllers can always reach desired, or *marked*, states. Some of these approaches require this property to be verified over the composition of the controlled subsystems – e.g., Wonham and Ramadge (1988); Queiroz and Cury (2000) – bringing a computational cost that, in the worst case, is of the same order as that of the monolithic approach (Ramadge and Wonham (1987)). There are works, such as that of Schmidt et al. (2008), in which such verification over the entire model is not required; this is done by establishing sufficient conditions for nonconflict over the local components, such that the verification process also becomes decentralised. Either way, if conflict is detected, it needs to be resolved somehow. One way is by designing a *coordinator* – i.e., an additional supervisor that coordinates the control actions of the local ones (see Wonham and Cai (2019)). Alternatively, if it is feasible to *communicate* the occurrence of events which are a priori nonshared from one supervisor to another – by the addition of sensors, for example – then it is possible to determine a set of external events that each local supervisor needs to observe to guarantee nonconflict – e.g. Su and Thistle (2006); Cai and Wonham (2016); Komenda and Masopust (2017), to cite a few.

Our method, on the other hand, guarantees *nonconflict by construction* without a coordinator or communication, being a suitable solution when communication is not desirable for security reasons, or is not physically or financially viable. To ensure nonconflict, we introduce here a new, targeted property called *relative unambiguity*, which is inspired by existing conditions for decomposability, such as relative observability from Cai et al. (2015) and locally nonblocking condition from Schmidt et al. (2008). In addition, we adopt the cooperative controllability setting by Su and Thistle (2006) to obtain *maximally permissive* local supervisors whenever the aforementioned property is immediately satisfied, i.e., does not need to be enforced.

Our framework is closely related to other iterative A/G-based synthesis approaches from the formal methods community, e.g. Apaza-Perez et al. (2020); Majumdar et al. (2020); Finkbeiner and Passing (2021). The particularity of our work in this context is the use of synchronised deterministic finite automata (DFA) over *finite* words as system models, compared to input/output ω-automata over *infinite* words. This brings interesting consequences. Firstly, to the best of our knowledge, there does not yet exist a sound assume-guarantee framework allowing ω-automata with arbitrary markings as contracts: existing work only uses fully marked (safety) automata for this purpose. Our method circumvents this problem, as the synchronisation of DFA ensures that liveness requirements, encoded by marked states in the involved automata, must be enforced by all subsystems at the same time. Secondly, synchronisation of automata leads to blocking situations in a decentralised setting – which requires special attention – while interacting input/output ω-automata never block.

2. PRELIMINARIES

In this section, we briefly introduce the notation and definitions relevant for this document. For a complete and more didactic coverage on supervisory control, see the textbooks by Cassandras and Lafortune (2021) and Wonham and Cai (2019).

2.1 Languages and Automata Basics

Strings. Let Σ be a *finite alphabet*, i.e., a finite set of symbols $\sigma \in \Sigma$ – which, when modelling a DES, represent the events of the system. The *Kleene-closure* Σ^* is the set of finite strings $s = \sigma_1\sigma_2\cdots\sigma_n$, with $n \in \mathbb{N}$ and $\sigma_i \in \Sigma$, including the *empty string* $\epsilon \in \Sigma^*$, with $\epsilon \notin \Sigma$. If, for two strings $s, r \in \Sigma^*$, there exists $t \in \Sigma^*$ such that $s = rt$, we say r is a *prefix* of s, and write $r \leq s$.

Languages. A *language* over Σ is a subset $L \subseteq \Sigma^*$. The *prefix* of a language $L \subseteq \Sigma^*$ is defined by $\overline{L} := \{r \in \Sigma^* \mid \exists\, s \in L : r \leq s\}$. The prefix operator is also referred to as the *prefix-closure*, and a language L is *closed* if $L = \overline{L}$. A language K is *relatively closed with respect to* L, or simply *L-closed*, if $K = \overline{K} \cap L$. The prefix operator distributes over arbitrary unions of languages. However, for the intersection of two languages L and M, we have $\overline{L \cap M} \subseteq \overline{L} \cap \overline{M}$. If equality holds, L and M are said to be *nonconflicting*.

Projections. Given two alphabets Σ and $\Sigma_i \subseteq \Sigma$, the *natural projection* $P_i : \Sigma^* \to \Sigma_i^*$ is defined recursively by $P_i(\epsilon) = \epsilon$ and for all $\sigma \in \Sigma$, $s \in \Sigma^*$ s.t. $P_i(s\sigma) = P_i(s)$ if $\sigma \notin \Sigma_i$ and $P_i(s)\sigma$ if $\sigma \in \Sigma_i$. We define the *inverse projection* $P_i^{-1} : \Sigma_i^* \to 2^{\Sigma^*}$ by $P_i^{-1}(s_i) = \{s \in \Sigma^* \mid P_i(s) = s_i\}$ for all $s_i \in \Sigma_i^*$. These definitions can be extended to languages, with $P_i : 2^{\Sigma^*} \to 2^{\Sigma_i^*}$ defined

such that, for any $L \subseteq \Sigma^*$, $P_i(L) = \{s_i \in \Sigma_i^* \mid \exists\, s \in L : P_i(s) = s_i\}$. In turn, $P_i^{-1} : 2^{\Sigma_i^*} \to 2^{\Sigma^*}$ is given by $P_i^{-1}(L_i) = \{s \in \Sigma^* \mid P_i(s) \in L_i\}$ for any $L_i \subseteq \Sigma_i^*$. The *synchronous product* of languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, where Σ_1 and Σ_2 are arbitrary alphabets, is defined as $L_1 \parallel L_2 := P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$.

Automata. A *deterministic finite automaton* (*automaton* for short) is a tuple $\mathbf{\Lambda} = (Q, \Sigma, \delta, q_0, Q_\mathrm{m})$, with finite *state set* Q, *initial state* $q_0 \in Q$, *marked states* $Q_\mathrm{m} \subseteq Q$, and the *deterministic transition function* $\delta : Q \times \Sigma \to Q$. This function is *partial*, i.e., $\delta(q, \sigma)$ is not necessarily defined for all $q \in Q$ and $\sigma \in \Sigma$; in the case it is, we say it is *total*. We identify δ with its common extension to the domain $Q \times \Sigma^*$. Let $\delta(q, s)!$ indicate that δ is defined for $q \in Q$ and $s \in \Sigma^*$; for all $q \in Q$, we have $\delta(q, \epsilon) = q$; for $s \in \Sigma^*$ and $\sigma \in \Sigma$, we have $\delta(q, s\sigma)!$, with $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma))$, if and only if $\delta(q, s)!$ and $\delta(\delta(q, s), \sigma))!$.

Reachability and nonblockingness. A state $q \in Q$ is *reachable* if there exists $s \in \Sigma^*$ such that $q = \delta(q_0, s)$, and it is *coreachable* if there exists $s \in \Sigma^*$ such that $\delta(q, s) \in Q_\mathrm{m}$. States which are not coreachable are also referred to as *blocking states*. If all reachable states in $\mathbf{\Lambda}$ are coreachable, then $\mathbf{\Lambda}$ is *nonblocking*. Moreover, $\mathbf{\Lambda}$ is called *reachable* (respectively *coreachable*) if all states are reachable (resp. coreachable), and $\mathbf{\Lambda}$ is called *trim* if it is reachable and coreachable. The *reachable* (resp. *coreachable*) subautomaton $\mathrm{Re}(\mathbf{\Lambda})$ (resp. $\mathrm{CoRe}(\mathbf{\Lambda})$) is obtained by eliminating all nonreachable (resp. blocking) states from $\mathbf{\Lambda}$ and, accordingly, adapting the transition function δ in the obvious way. The *trim subautomaton* $\mathrm{Trim}(\mathbf{\Lambda})$ is given by $\mathrm{Re}(\mathrm{CoRe}(\mathbf{\Lambda})) = \mathrm{CoRe}(\mathrm{Re}(\mathbf{\Lambda}))$.

Semantics. With the automaton $\mathbf{\Lambda} = (Q, \Sigma, \delta, q_0, Q_\mathrm{m})$, we associate the *generated language* $\mathcal{L}(\mathbf{\Lambda}) := \{\, s \in \Sigma^* \mid \delta(q_0, s)! \,\}$ and the *marked language* $\mathcal{L}_m(\mathbf{\Lambda}) := \{\, s \in \Sigma^* \mid \delta(q_0, s) \in Q_\mathrm{m} \,\}$. We say these two languages are the *semantics* of $\mathbf{\Lambda}$, and that $\mathbf{\Lambda}$ *recognizes*, or *accepts*, the language $\mathcal{L}_m(\mathbf{\Lambda})$. Moreover, we say any two automata are *equivalent* if their semantics coincide. Note that $\mathbf{\Lambda}$ is nonblocking if and only if $\mathcal{L}(\mathbf{\Lambda}) = \overline{\mathcal{L}_m(\mathbf{\Lambda})}$. To simplify notation, we use boldface characters, e.g. $\mathbf{\Lambda}$, to denote an automaton, and the corresponding normal character, e.g. Λ, for its marked language.

Composition. Given two automata $\mathbf{\Lambda}_i = (Q_i, \Sigma_i, \delta_i, q_{i_0}, Q_{i\mathrm{m}})$, $i \in \{1, 2\}$, their *synchronous composition* $\mathbf{\Lambda}_1 \parallel \mathbf{\Lambda}_2$ is defined as the standard synchronous product of finite automata, that is, events $\sigma \in \Sigma^\mathrm{s} = \Sigma_1 \cap \Sigma_2$ require synchronisation of the corresponding transitions, while transitions over nonshared events $\sigma_i \in \Sigma_i \backslash \Sigma^\mathrm{s}$ are not restricted by the product. We use here the same symbol to denote the synchronous product of languages and the synchronous composition of automata; thus, $\mathcal{L}(\mathbf{\Lambda}_1 \parallel \mathbf{\Lambda}_2) = \mathcal{L}(\mathbf{\Lambda}_1) \parallel \mathcal{L}(\mathbf{\Lambda}_2)$ and $\mathcal{L}_m(\mathbf{\Lambda}_1 \parallel \mathbf{\Lambda}_2) = \Lambda_1 \parallel \Lambda_2$.

2.2 Supervisory Control

Plant Model. A plant is a system to be supervised, and it is modelled as an automaton $\mathbf{M} = (Q, \Sigma, \delta, q_0, Q_\mathrm{m})$ whose alphabet Σ can be partitioned as $\Sigma = \Sigma_\mathrm{c} \dot\cup \Sigma_\mathrm{uc}$, where Σ_c is the set of *controllable* events – those that can be prevented from happening, or disabled – and Σ_uc is the set of *uncontrollable* ones – that cannot be disabled.

Specification. A *specification* for a plant $\mathbf{M}$ over the alphabet Σ is modelled by a trim automaton $\mathbf{E}$ over $\Sigma' \subseteq \Sigma$. The purpose of $\mathbf{E}$ is to model merely which sequences of events in Σ' are allowed to occur in the closed-loop behaviour, without taking into account the sequences that are indeed possible in the free (open-loop) behaviour of $\mathbf{M}$. A plant can be composed with multiple specifications, and the resulting automaton is then called its *desired behaviour*. It is useful to design the specifications such that their transition function is total; this is done by adding a blocking state reached by all sequences of events undesirable in the controlled dynamics. Here, we only consider specifications where this is the case, and where all the states but the blocking one are marked. The desired behaviour is, then, a blocking automaton $\mathbf{K}$ whose semantics are contained in the semantics of the plant, and whose marked language is M-closed. We call $\mathbf{K}$ a *plantified specification*. Please note that *specification* and *plantified specification* are not interchangeable terms.

Controllability. A language L over Σ is *controllable* with respect to $\mathcal{L}(\mathbf{M})$ and Σ_uc if, for all $\sigma \in \Sigma_\mathrm{uc}$, $s \in \overline{L} \wedge s\sigma \in \mathcal{L}(\mathbf{M}) \;\Rightarrow\; s\sigma \in \overline{L}$. Define the set $\mathcal{C}(L) := \{L' \subseteq L \mid L' \text{ is controllable w.r.t. } \mathcal{L}(\mathbf{M}) \text{ and } \Sigma_\mathrm{uc}\}$, whether L is controllable or not. This set is nonempty, since the empty language is trivially controllable. As controllability is closed under union of languages, it can be shown that the supremum element of $\mathcal{C}(L)$, denoted $\sup\mathcal{C}(L)$, is given by $\bigcup_{L' \in \mathcal{C}(L)} L'$ and is controllable, i.e., belongs to $\mathcal{C}(L)$.

Observability. A language L over Σ is *observable* with respect to $\mathcal{L}(\mathbf{M})$, Σ' and $P_\mathrm{s} : \Sigma^* \to (\Sigma^\mathrm{s})^*$, where $\Sigma^\mathrm{s}, \Sigma' \subseteq \Sigma$, if, for any $\sigma \in \Sigma'$ and $s, s' \in \overline{L}$ such that $P_\mathrm{s}(s) = P_\mathrm{s}(s')$, we have that $\left[s\sigma \in \overline{L} \text{ and } s'\sigma \in \mathcal{L}(\mathbf{M})\right] \Rightarrow s'\sigma \in \overline{L}$.

Supervisor. A *supervisor* for a plant $\mathbf{M}$ with alphabet $\Sigma := \Sigma_\mathrm{c} \dot\cup \Sigma_\mathrm{uc}$ is a mapping $f : \mathcal{L}(\mathbf{M}) \to \Gamma$, where $\Gamma := \{\gamma \subseteq \Sigma \mid \Sigma_\mathrm{uc} \subseteq \gamma\} \subseteq 2^\Sigma$ is the set of *control patterns*. Let us denote by $f/\mathbf{M}$ the plant $\mathbf{M}$ under supervision of f. The generated language of $f/\mathbf{M}$ is defined recursively such that $\epsilon \in \mathcal{L}(f/\mathbf{M})$ and, for all $s \in \Sigma^*$ and $\sigma \in \Sigma$, $s\sigma \in \mathcal{L}(f/\mathbf{M})$ iff (i) $s \in \mathcal{L}(f/\mathbf{M})$, (ii) $s\sigma \in \mathcal{L}(\mathbf{M})$, and (iii) $\sigma \in f(s)$. This induces the marked language $\mathcal{L}_m(f/\mathbf{M}) := \mathcal{L}(f/\mathbf{M}) \cap M$, which is controllable by definition. Note that $\mathcal{L}(f/\mathbf{M})$ is closed and $\mathcal{L}(f/\mathbf{M}) \subseteq \mathcal{L}(\mathbf{M})$. We call f nonblocking if $\mathcal{L}(f/\mathbf{M}) = \overline{\mathcal{L}_m(f/\mathbf{M})}$. In this case, the closed loop can be represented by a trim automaton $\mathbf{S}$ that accepts $\mathcal{L}_m(f/\mathbf{M})$; we then say that $\mathbf{S}$ *realises* $f/\mathbf{M}$ and denote this by $\mathbf{S} \sim f$.

Supervisor Synthesis Problem. Given a plant $\mathbf{M}$ and a plantified specification $\mathbf{K}$ over an alphabet $\Sigma := \Sigma_\mathrm{c} \dot\cup \Sigma_\mathrm{uc}$, the control problem is to design a maximally permissive supervisor f that respects the specifications – i.e., $\mathcal{L}_m(f/\mathbf{M}) = \sup\mathcal{C}(K)$, where K is the language accepted by $\mathbf{K}$ – and that imposes a nonblocking closed-loop behaviour – that is, $\mathcal{L}(f/\mathbf{M}) = \overline{\mathcal{L}_m(f/\mathbf{M})}$.

Supervisor Computation. There exist many different procedures to algorithmically solve the stated synthesis problem. In particular, from a plantified specification $\mathbf{K}$ it is possible to compute a trim automaton $\mathbf{S} \sim f$ only by manipulating the first automaton, without the plant $\mathbf{M}$ as an additional input – see (Cassandras and Lafortune,

2021, p.186) for details. With that in mind, let us define the function SYNTH.

Definition 1. For a plant $\mathbf{M}$ and a plantified specification $\mathbf{K}$ over an alphabet $\Sigma := \Sigma_{\mathrm{c}} \dot{\cup} \Sigma_{\mathrm{uc}}$, we define the function SYNTH such that $\textsc{Synth}(\mathbf{K}) = \mathbf{S}$, where $\mathbf{S}$ is a trim automaton that accepts the language $\sup\mathcal{C}(K)$ with respect to $\mathbf{M}$ and Σ_{uc}.

3. ASSUME-GUARANTEE SYNTHESIS

This section introduces the decentralised supervisor synthesis problem we tackle in this report and formalises the concept of assume-guarantee synthesis in this particular setting. In order to simplify notation, we only consider the special case of *two* decentralised processes. However, as all computations for synthesis are done fully locally, an extension to other plant structures is straightforward. Please note that the proofs of all the results presented in this document can be found in Appendix A.

3.1 Decentralised Supervisor Synthesis Problem

We consider a system composed of two processes, both modelled as plant automata $\mathbf{M}_i = (Q_i, \Sigma_i, \delta_i, q_{i_0}, Q_{i_{\mathrm{m}}})$, $i \in \{1, 2\}$. Each local alphabet is partitioned into $\Sigma_i = \Sigma_i^{\ell} \dot{\cup} \Sigma^{\mathrm{s}}$, where Σ^{s} is the set of *shared* events – i.e., $\Sigma^{\mathrm{s}} = \Sigma_1 \cap \Sigma_2$ – and Σ_i^{ℓ} is the set of *local* events exclusive to $\mathbf{M}_i$, i.e., $\Sigma_i^{\ell} = \Sigma_i \setminus \Sigma^{\mathrm{s}}$. We assume that Σ_i^{ℓ} may contain controllable and uncontrollable events, meaning $\Sigma_i^{\ell} = \Sigma_{i,\mathrm{uc}}^{\ell} \dot{\cup} \Sigma_{i,\mathrm{c}}^{\ell}$. To further simplify the presentation, we assume that all shared events are controllable by at least one subsystem [1]. Yet we do *not* require both processes to agree on the controllability status of shared events, that is, we have $\Sigma^{\mathrm{s}} = \Sigma_{1,\mathrm{c}}^{\mathrm{s}} \cup \Sigma_{2,\mathrm{c}}^{\mathrm{s}}$, where $\Sigma_{i,\mathrm{c}}^{\mathrm{s}} := \Sigma_{i,\mathrm{c}} \setminus \Sigma_i^{\ell}$ and $\Sigma_{i,\mathrm{c}}$ is the set of all events controllable by $\mathbf{M}_i$. Finally, each process is equipped with a plantified specification $\mathbf{K}_i$ over the alphabet Σ_i. To simplify notation throughout the document, we use the convention that the indices i, j are understood from the context such that $i \neq j$ for all $i, j \in \{1, 2\}$.

3.2 Contract-based Supervisor Synthesis.

The fundamental feature that assume-guarantee synthesis exploits is the use of *contracts* to ensure the necessary *cooperation* between decentralised components. Since each local process can observe the other's dynamics solely by the occurrence of events they share, contracts are defined by automata over such events and they represent both what one local supervisor expects from and what it promises to the other controlled process in terms of disabling those events. In our motivating example in Sec. 1.1, if we consider each manufacturing line as a subsystem, the type of workpiece arriving in a line through a shared buffer is dependent on decisions made locally in the other line; in this case, a contract specifies – in terms of a language over Σ^{s} – which sequences of workpieces in the shared buffer allow both lines to fulfil their specification.

Assume-Guarantee Contracts. Towards a formalisation of this insight, we start by modelling a contract for the subsystem $\mathbf{M}_i$ as a tuple of automata $(\mathbf{A}_i, \mathbf{G}_i)$ over the shared alphabet Σ^{s}. The automaton $\mathbf{G}_i$ represents the *guarantee* the subsystem needs to provide for the rest of the plant. This guarantee is an additional local *specification*, in the sense defined in Sec. 2.2: it solely informs which sequences of events – in this case, shared ones – are allowed to occur in the local closed-loop behaviour, disregarding which ones are actually generated by $\mathbf{M}_i$.

The automaton $\mathbf{A}_i$ is the *assumption*. It models the closed-loop behaviour of the rest of the plant as perceived by subsystem i. Thus, the local supervisor f_i does not enforce over its subsystem the restrictions that are needed to satisfy its local specifications but that are assumed to be *already imposed* by the rest of the plant – since all controlled subsystems synchronise over shared events in the global dynamics.

For the local specifications and also the guarantee to be satisfied, the contracts should therefore be obtained in such a way that a local supervisor does not assume that more restrictions are being imposed by the other closed-loop subsystem than the latter actually guarantees. We state this by writing that their contracts are *compatible* if $\mathcal{L}(\mathbf{G}_j) \subseteq \mathcal{L}(\mathbf{A}_i)$ and $G_j \subseteq A_i$. Finally, by interpreting $\mathbf{G}_i$ and $\mathbf{A}_i$ respectively as an extra specification and as an extra plant model, we see that the single automaton $\mathbf{C}_i = \mathbf{A}_i \parallel \mathbf{G}_i$ captures all the information needed for the contract [2]. This leads to the following contract-based supervisor synthesis problem.

Definition 2. Given a plant $\mathbf{M}$ with a plantified specification $\mathbf{K}$ over Σ and a contract $\mathbf{C}$ over $\Sigma' \subseteq \Sigma$, we define $\mathbf{S} := \textsc{Synth}(\mathbf{K} \parallel \mathbf{C})$ as the contract-based supervisor automaton for $\mathbf{M}$ with respect to $\mathbf{K}$ and $\mathbf{C}$, inducing the contract-based supervisor f such that $\mathbf{S} \sim f$.

Thus, it directly follows from the definition of SYNTH that $\mathcal{L}_m(f/\mathbf{M}) = S = \sup\mathcal{C}(K \parallel C)$.

Maximal Permissiveness. For we are interested in designing local supervisors such that the global closed-loop behaviour is maximally permissive, the contracts should not only be compatible as defined before, but their guarantees should be the least restrictive possible, i.e., $\mathcal{L}(\mathbf{G}_j) = \mathcal{L}(\mathbf{A}_i)$ and $G_j = A_i$. This implies that the contract automata $\mathbf{C}_1$ and $\mathbf{C}_2$ have the same semantics, that is, they are *equivalent*.

Outline. In Sec. 4, we introduce an algorithm for contract negotiation which computes maximally permissive local supervisors f_i in a *decentralised manner*. Then, in Sec. 5, we show how this algorithm must be adapted to ensure that the marked language of the global behaviour of the system in closed-loop with these supervisors is nonblocking. As expected, this might sacrifice optimality, i.e., this language may become a strict subset of $\sup\mathcal{C}(K_1 \parallel K_2)$ with respect to $\mathcal{L}(\mathbf{M}_1 \parallel \mathbf{M}_2)$ and $\Sigma_{1,\mathrm{uc}}^{\ell} \cup \Sigma_{2,\mathrm{uc}}^{\ell}$.

4. CONTRACT NEGOTIATION

This section introduces a negotiation framework to compute maximally permissive supervisors f_i for each sub-

[1] In principle, the set of shared events could contain events which are uncontrollable to both systems; this case requires special treatment.

[2] This is conceptually similar to the plantified specification $\mathbf{K}_i$ capturing all the information needed for the supervisory synthesis problem for plant model $\mathbf{M}_i$ with respect to specification $\mathbf{E}_i$.

system in a *decentralised* manner. This is achieved by a *local* iterative refinement of the plantified specification automata $\mathbf{K}_i$. At each iteration and for each subsystem, states and transitions are removed from $\mathbf{K}_i$ to prevent blocking, ensure controllability, and to satisfy the guarantees $\mathbf{G}_i$ for the other subsystem, while relying on the latter to enforce the assumptions $\mathbf{A}_i$. The negotiation takes place in a way that avoids the plant to be overly restricted, resulting in maximally permissive supervisors. Yet, as we discuss in the next section, they may still be conflicting, which can be both tested and fixed fully locally, even though the latter may cost the maximally permissive trait.

4.1 Cooperative Supervisor Synthesis

Let us recall that, in the setting we consider here, the local systems may not agree on the controllability status of their shared events. Inspired by Su and Thistle (2006), the idea is therefore that their local supervisors cooperate in such a way as to assist each other in disabling shared events that are uncontrollable for one, but controllable for the other – otherwise, control actions designed locally may over-restrict the behaviour of the plant.

To understand why *cooperative* local supervisor synthesis is needed to preserve maximal permissiveness, note that, for each subsystem $\mathbf{M}_i$, its set of uncontrollable events is $\Sigma^{\ell}_{i,\mathrm{uc}} \dot{\cup} (\Sigma^{\mathrm{s}} \backslash \Sigma^{\mathrm{s}}_{i,\mathrm{c}})$. If we consider controllability with respect to those events while computing $\textsc{Synth}(\mathbf{K}_i)$, states from $\mathbf{K}_i$ where the only uncontrollable events being disabled are precisely the ones the other system can control are eliminated, since they disrespect the controllability property. However, theses states would not be removed in the monolithic approach, unless they were blocking or unreachable states; the reason is that events controllable either by one or by the other subsystem would be considered as controllable by the global plant. To illustrate, suppose a trivial example where both subsystems have exactly the same behaviour and already respect their specification – that is, $\mathbf{M}_1$, $\mathbf{M}_2$, $\mathbf{K}_1$ and $\mathbf{K}_2$ all have the same semantics. Both subsystems have full knowledge about each other without further exchange of contracts, as all events are shared – which also implies that all events are controlled by some subsystem. Even so, without cooperation it may not be possible to locally design optimal supervisors, as depicted in Figure 2.

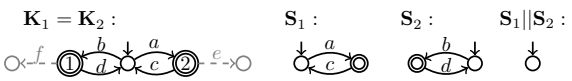

Fig. 2. Lack of cooperation may cause over-restrictive behaviour. Consider that $b, e \in \Sigma^{\mathrm{s}}_{1,\mathrm{c}} \backslash \Sigma^{\mathrm{s}}_{2,\mathrm{c}}$ and $a, f \in \Sigma^{\mathrm{s}}_{2,\mathrm{c}} \backslash \Sigma^{\mathrm{s}}_{1,\mathrm{c}}$; then, the elimination of blocking states in $\mathbf{K}_{1,2}$, as depicted in gray colour, generates a controllability problem in different states for each subsystem, which in turn have to be removed as well; thus, the languages $\sup\mathcal{C}(K_i)$ – recognised by the automata $\mathbf{S}_i$ – are conflicting. If cooperation was assumed, states 1 and 2 would be preserved, and $\mathbf{S}_1 \parallel \mathbf{S}_2$ would be equivalent to the automaton $\mathbf{K}_{1,2}$ apart from its blocking states in gray.

Formally, this cooperation requires to perform Synth of $\mathbf{K}_i$ with respect to the uncontrollable event set $\Sigma^{\ell}_{i,\mathrm{uc}}$, instead of the actual locally uncontrollable event set $\Sigma_{i,\mathrm{uc}} = \Sigma^{\ell}_{i,\mathrm{uc}} \dot{\cup} (\Sigma^{\mathrm{s}} \backslash \Sigma^{\mathrm{s}}_{i,\mathrm{c}})$. In order to avoid confusion, let us define a cooperative version of this function below.

Definition 3. For the plantified specification $\mathbf{K}_i$, we define the function CSynth such that $\textsc{CSynth}(\mathbf{K}_i) = \mathbf{S}_i$, where $\mathbf{S}_i$ is a trim automaton that accepts the language $\sup\mathcal{C}(K_i)$ with respect to $\mathbf{M}_i$ and $\Sigma^{\ell}_{i,\mathrm{uc}}$.

4.2 Contract Extraction

Consider the automaton $\mathbf{S}_i := \textsc{CSynth}(\mathbf{K}_i)$. This automaton implicitly disables shared events that are not controllable in $\mathbf{M}_i$ but are in $\mathbf{M}_j$, presuming that the supervisor of the latter will assist that of the former in disabling them. At the same time, $\mathbf{S}_i$ also disables locally controllable events to achieve its own local specifications, ensure controllability and local nonblockingness. This, in turn, may also restrict the occurrence of shared events in $\mathbf{S}_j$ due to the synchronisation in the global behaviour.

Motivated by the discussion in Sec. 3.2, a natural choice for the first draft of a contract automaton $\mathbf{C}_i$ – that is, the contract generated by $\mathbf{M}_j$ to be used by $\mathbf{M}_i$ during the next round of negotiation – is the *observer* automaton of $\mathbf{S}_j$ over Σ^{s}, defined as follows.

Definition 4. (Cassandras and Lafortune (2021)). For an automaton $\mathbf{\Lambda} = (Q, \Sigma, \delta, q_0, Q_{\mathrm{m}})$, an alphabet $\Sigma' \subseteq \Sigma$, and a natural projection $P : 2^{\Sigma^*} \to 2^{(\Sigma')^*}$, we define the *observer* automaton $\mathbf{O}_{\mathbf{\Lambda}} := (Q', \Sigma', \delta', q'_0, Q'_{\mathrm{m}})$ whose elements are as follows. For each $p \in P(\mathcal{L}(\mathbf{\Lambda}))$, there is a state $q' \in Q' \subseteq 2^Q$ such that

$$q' = \{q \in Q \mid \exists s \in \mathcal{L}(\mathbf{\Lambda}) : P(s) = p \text{ and } q = \delta(q_0, s)\}.$$

Further, $\delta'(q'_1, \sigma) = q'_2$ if there exists $q_1 \in q'_1$ and $q_2 \in q'_2$ such that $\delta(q_1, \sigma) = q_2$; otherwise, $\delta'(q'_1, \sigma)$ is undefined. The initial state $q'_0 \in Q'$ is such that $q_0 \in q'_0$, and the set of marked states is $Q'_{\mathrm{m}} = \{q' \in Q' \mid \exists q \in q' : q \in Q_{\mathrm{m}}\}$.

By defining $\mathbf{C}_i := \mathbf{O}_{\mathbf{S}_j}$, the contract generates and accepts the languages $P_{j\mathrm{s}}(\mathcal{L}(\mathbf{S}_j))$ and $P_{j\mathrm{s}}(S_j)$, respectively, considering the natural projection $P_{j\mathrm{s}} : 2^{\Sigma^*_j} \to 2^{(\Sigma^{\mathrm{s}})^*}$.

4.3 Negotiation

Based on the foregoing discussion, we propose the following iterative procedure for the negotiation of contracts. Initially, we compute $\mathbf{S}^0_i := \textsc{CSynth}(\mathbf{K}_i)$ and extract the first contract drafts $\mathbf{C}^0_i$. Next, we compute new supervisors restricted to the latest drafts of contracts, namely, $\mathbf{S}^0_i \parallel \mathbf{C}^0_i$; however, as this composition might introduce new controllability or blocking problems, we perform CSynth again. Inducing this argument for an arbitrary step $k > 0$, we have that $\mathbf{S}^k_i := \textsc{CSynth}(\mathbf{S}^{k-1}_i \parallel \mathbf{C}^{k-1}_i)$, which generates a new draft of contract given by $\mathbf{C}^k_i := \mathbf{O}_{\mathbf{S}^k_j}$.

The entire procedure, called Negotiation, is detailed in Algorithm 1. It iteratively refines $\mathbf{S}_i$ and $\mathbf{C}_i$ until $\mathbf{S}^k_i = \mathbf{S}^{k-1}_i \parallel \mathbf{C}^{k-1}_i$, implying that no new controllability or blocking issues need to be solved. Thus, upon termination we have that the automata $\mathbf{S}_i$ are trim and that $C_2 = P_{1\mathrm{s}}(S_1) = P_{2\mathrm{s}}(S_2) = C_1$, that is, the resulting contracts

Algorithm 1 NEGOTIATION

Require: Automata $\mathbf{S}_1^{\text{init}}$ and $\mathbf{S}_2^{\text{init}}$
$\mathbf{S}_1 \leftarrow \text{CSYNTH}(\mathbf{S}_1^{\text{init}})$ and $\mathbf{S}_2 \leftarrow \text{CSYNTH}(\mathbf{S}_2^{\text{init}})$
$\mathbf{C}_1 \leftarrow \mathbf{O}_{\mathbf{S}_2}$ and $\mathbf{C}_2 \leftarrow \mathbf{O}_{\mathbf{S}_1}$
cond $\leftarrow$ `True`
while *cond* **do**
  if $\mathbf{S}_1 \not\equiv (\mathbf{S}_1 \parallel \mathbf{C}_1)$ or $\mathbf{S}_2 \not\equiv (\mathbf{S}_2 \parallel \mathbf{C}_2)$ **then**
    for $i \in \{1, 2\}$ **do**
      if $\mathbf{S}_i \not\equiv \mathbf{S}_i \parallel \mathbf{C}_i$ **then**
        $\mathbf{S}_i \leftarrow \text{CSYNTH}(\mathbf{S}_i \parallel \mathbf{C}_i)$ and $\mathbf{C}_j \leftarrow \mathbf{O}_{\mathbf{S}_i}$ $(j \neq i)$
      end if
    end for
  else *cond* $\leftarrow$ `False`
  end if
end while
return $\mathbf{S}_1$ and $\mathbf{S}_2$

are equivalent. Furthermore, the algorithm does not overly restrict the behaviour of the system, as stated in the theorem below.

Theorem 1. Consider a plant composed of two subsystems $\mathbf{M}_i$ and their corresponding plantified specifications $\mathbf{K}_i$, as described in Sec. 3.1. Let $\mathbf{S}_i$ be the outputs of Algorithm 1 for the inputs $\mathbf{K}_i$, namely, $(\mathbf{S}_1, \mathbf{S}_2) = \text{NEGOTIATION}(\mathbf{K}_1, \mathbf{K}_2)$. Then, we have that

$$\sup\mathcal{C}(S_1 \parallel S_2) = \sup\mathcal{C}(K_1 \parallel K_2),$$

where $\sup\mathcal{C}$ is taken with respect to $\mathcal{L}(\mathbf{M}_1 \parallel \mathbf{M}_2)$ and $\Sigma^{\ell}_{1,\text{uc}} \cup \Sigma^{\ell}_{2,\text{uc}}$.

Since the function NEGOTIATION refines the automata given as inputs, we have that $S_i \subseteq K_i$; hence, it is trivial that $\sup\mathcal{C}(S_1 \parallel S_2) \subseteq \sup\mathcal{C}(K_1 \parallel K_2)$. The result of the theorem above shows the set inclusion in the other direction, proving that the iterative refinements of the inputs do not remove any more states and transitions than the necessary to obtain $\sup\mathcal{C}(K_1 \parallel K_2)$ – note that they may remove less, though, in which case we further need to guarantee that the resulting composed system is not blocking, as discussed later in Sec. 5. However, due to the definition of CSYNTH, each fixed point $\mathbf{S}_i$ is locally nonblocking.

4.4 Supervisor extraction

As $\mathbf{S}_i$ is a *cooperative* supervisor, it remains to extract a local supervisor f_i which only disables locally controllable events in the set $\Sigma_{i,\text{c}}$, as defined below.

Definition 5. Given the premisses of Thm. 1 and if $\mathbf{S}_i$ are nonempty automata, we define each local supervisor $f_i : \mathcal{L}(\mathbf{M}_i) \to \Gamma_i$, with $\Gamma_i \subseteq 2^{\Sigma_i}$, such that $f_i(s) = \{\sigma \mid s\sigma \in \overline{S_i}\} \cup \Sigma^{\text{s}} \backslash \Sigma^{\text{s}}_{i,\text{c}}$.

The resulting supervisors f_i are less restrictive than $\mathbf{S}_i$, but if the proposed cooperation can be assured, then their composed closed-loop behaviour is the same. The simple example depicted in Fig. **??** illustrates why cooperation may fail. However, an observability condition is sufficient to eliminate the possibility of such scenario and, hence, guarantee the local controllers indeed cooperate in disabling shared events, as formalised in the following lemma.

Lemma 1. In the context of Definition 5, the following equalities hold if and only if S_i are observable with respect to $\mathcal{L}(\mathbf{M}_i)$, $\Sigma^{\text{s}} \backslash \Sigma^{\text{s}}_{i,\text{c}}$ and $P_{i\text{s}} : \Sigma_i^* \to (\Sigma^{\text{s}})^*$:

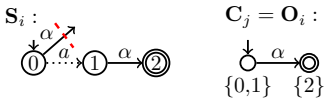

Fig. 3. Scenario where cooperation fails. The event a is local, while α is shared but locally uncontrollable for $\mathbf{S}_i$ (and controllable for $\mathbf{S}_j$); the transition with *alpha* from state 0 needs to be disabled and f_i cannot do this, so cooperation is needed; however, since a transition by the same event is enabled in state 1, it is not possible to represent that disabling in the observer of $\mathbf{S}_i$, i.e., in contract $\mathbf{C}_j$.

(1) $\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2) = S_1 \parallel S_2$ and
(2) $\mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2) = \overline{S_1} \parallel \overline{S_2}$.

From this lemma, the next result immediately holds.

Corollary 1. In the context of Definition 5, the global behaviour of the plant in closed loop with the supervisors f_i is nonblocking, that is,

$$\mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2) = \overline{\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2)},$$

if and only if S_1 and S_2 are nonconflicting and S_i observable with respect to $\mathcal{L}(\mathbf{M}_i)$, $\Sigma^{\text{s}} \backslash \Sigma^{\text{s}}_{i,\text{c}}$ and $P_{i\text{s}} : \Sigma_i^* \to (\Sigma^{\text{s}})^*$.

Furthermore, from Thm. 1 we immediately get the main result of this section, as follows.

Theorem 2. Given the premisses of Thm. 1, if S_1 and S_2 are nonconflicting we have that

$$S_1 \parallel S_2 = \sup\mathcal{C}(K_1 \parallel K_2),$$

where $\sup\mathcal{C}$ is taken with respect to $\mathcal{L}(\mathbf{M}_1 \parallel \mathbf{M}_2)$ and $\Sigma^{\ell}_{1,\text{uc}} \cup \Sigma^{\ell}_{2,\text{uc}}$.

5. ASSURING NONCONFLICTING LOCAL SUPERVISORS

This section shows how local supervisors resulting from negotiation may have to be further restricted to ensure a nonblocking behaviour of the resulting global closed-loop system. As we aim at computing necessary restrictions fully locally, we might sacrifice maximal permissiveness.

5.1 Motivating Example

Consider the example in Fig. 4, where $\mathbf{S}_1$ and $\mathbf{S}_2$ are nonblocking automata with shared alphabet $\Sigma^{\text{s}} = \{\alpha, \beta, \theta\}$; assume controllability (with cooperation) is not being disrespected. They satisfy the fixed point property for NEGOTIATION; however, their composed behaviour is blocking.

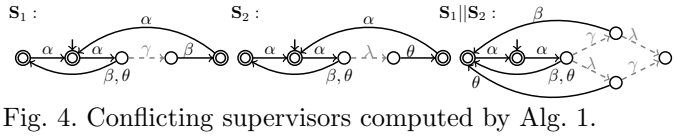

Fig. 4. Conflicting supervisors computed by Alg. 1.

To understand the reason behind this conflict, note that the strings α and $\alpha\lambda$ in $\overline{S_2}$ look the same to $\mathbf{S}_1$, as $P_{2\text{s}}(\alpha) = P_{2\text{s}}(\alpha\lambda) = \alpha$. This conceals the fact that β can occur in $\mathbf{S}_2$ after α, but not after $\alpha\lambda$. Analogously, α and $\alpha\gamma$ look the same to $\mathbf{S}_2$, hiding the fact that θ can occur in $\overline{S_1}$ after α, but not after $\alpha\gamma$. Thus, strings like $\alpha\gamma\lambda$ or $\alpha\lambda\gamma$ can occur in $\overline{S_1} \parallel \overline{S_2}$ and lead to blocking states.

5.2 The Unambiguity Property

In order to ensure nonconflict in the global behaviour of the plant whose subsystems are in closed loop with their respective local supervisor – that is, to prevent situations as the one just described in the last subsection – it is sufficient to guarantee that each controlled subsystem satisfies the following property, here named *unambiguity*.

Definition 6. The language S_i is *unambiguous* with respect to Σ^{s} and the natural projection $P_{i\mathrm{s}} : \Sigma_i^* \to (\Sigma^{\mathrm{s}})^*$ if, for any $s, s' \in \overline{S_i}$ such that $P_{i\mathrm{s}}(s) = P_{i\mathrm{s}}(s')$,

(1) $(\forall \sigma \in \Sigma^{\mathrm{s}})\ s\sigma \in \overline{S_i} \ \Rightarrow (\exists s'' \in \Sigma_i^*)$
$s' \leq s''$, $P_{i\mathrm{s}}(s'') = P_{i\mathrm{s}}(s')$ and $s''\sigma \in \overline{S_i}$;
(2) $s \in S_i \Rightarrow (\exists s'' \in \Sigma_i^*)$
$s'' \leq s'$or $s' < s''$, $P_{i\mathrm{s}}(s'') = P_{i\mathrm{s}}(s')$ and $s'' \in S_i$.

The reasons for the name are the following. A string s_s over the shared alphabet Σ^{s} can be the projection of different strings over a local alphabet Σ_i; some may allow a shared event σ to eventually happen, possibly after a suffix string of nonshared events, while others may not; thus, this ambiguity of s_s makes it possible for conflict to happen – e.g., consider as s_s the projection α in the example from Fig. 4. This scenario is prevented by condition (1), which also appears in Schmidt et al. (2008), where it is called *locally nonblocking* condition – in our context, though, the supervisors are designed locally and can observe the entire local alphabet, i.e., they can guarantee local nonblockingness without further assumptions.

Now, assume each automaton $\mathbf{S}_i$ satisfies condition (1), but not the (2); then, their marking is ambiguous and, although all marked states of each automaton can be reached in the composed behaviour, conflict may still arise if for some path they are never reached simultaneously in both automata.

To better explain this notion, let us take a look at the Fig. 5, where $\mathbf{S}_i$ are trim automata and $\Sigma^{\mathrm{s}} = \{\alpha, \beta\}$; assume controllability and the observability condition for cooperation are not being disrespected. Let us consider here two different markings. In the first one, indicated by the colours black and red, only the states 3 and 5 are marked in $\mathbf{S}_1$, the states $\{1, 2, 5\}$ and $\{3, 4\}$ in $\mathbf{O}_1$, the states 2 and 4 in $\mathbf{S}_2$, and the states $\{1, 4\}$ and $\{2, 3\}$ in $\mathbf{O}_2$. The second marking is shown by the colours black and blue, so the only marked states in $\mathbf{S}_1$ are 2 and 5, in $\mathbf{O}_1$ the state $\{1, 2, 5\}$, in $\mathbf{S}_2$, the states 1 and 4, and in $\mathbf{O}_2$, the state $\{1, 4\}$.

As in the example from the previous subsection, the supervisors from Fig. 5 satisfy, in both marking scenarios, the fixed point property for NEGOTIATION, i.e., their semantics are the same under the projection over shared events; moreover, the unambiguity condition (1) is also satisfied. For the first marking, though, the condition (2) is not. The reason is that from the perspective of one subsystem, the only marking that can be observed from the other one is through its observer. Then, for instance, we have that the sequence α is in the marked language of the observer, while $\mathbf{S}_1$ could be in the state 1 or 2, and neither is marked; moreover, if a β is observed after α, $\mathbf{S}_1$ could be in the state 3 or 4, and again it is not possible to know just by looking at $\mathbf{O}_1$ whether a marked state was actually reached in $\mathbf{S}_1$. This leads to conflict in this example simply because the markings of $\mathbf{S}_i$ do not synchronise; although marked states in each automaton can always be reached even in the composed behaviour, both automata are never in a marked state at the same time. Indeed, in the global behaviour $\mathbf{S}_2$ may get locked in the loop between the states 3 and 4, while $\mathbf{S}_1$ is locked in the cycle with its states 1, 2 and 3, which indicates that S_i are conflicting.

Now consider the example with the second marking. Then, note that there is no conflict and that unambiguity is satisfied – e.g., states 1, 2 and 5 from $\mathbf{S}_1$ correspond to the same state in $\mathbf{O}_1$, and although 1 is the only one not marked, it can reach 2 by local event b. It is also worth noting that condition (2) from unambiguity is not equivalent to the *marked string acceptance* from Schmidt et al. (2008); indeed, in the second scenario from this example, the former is satisfied while the latter is not – because state 1 in $\mathbf{S}_1$ can be followed by β and it is not marked.

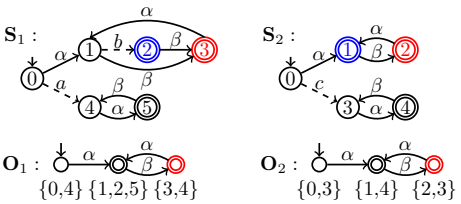

Fig. 5. Another example of conflicting supervisors computed by Alg. 1; each one of them does not satisfy condition (2) from the unambiguity property.

Based on the intuition from the foregoing discussion, we state the following result.

Proposition 1. For trim automata $\mathbf{S}_1$ and $\mathbf{S}_2$ as in Thm. 1, if their marked languages S_1 and S_2 are unambiguous with respect to Σ^{s} and the natural projection $P_{i\mathrm{s}} : \Sigma_i^* \to (\Sigma^{\mathrm{s}})^*$, then these languages are nonconflicting.

5.3 Locally Enforcing Unambiguity

Note that unambiguity is not retained under union of languages and therefore a supremal unambiguous sublanguage of S_i may not exist. The same is true for the observability condition stated on Lem. 1, necessary for the cooperation between the supervisors. Since we do not want to just check whether each subsystem satisfies such properties, but rather enforce them over the languages S_i, we are interested in a supremal sublanguage of S_i that satisfies a sufficient condition for nonconflict and also for cooperation. Inspired by the concept of relative observability (Cai et al. (2015)), we introduce *relative unambiguity*.

Definition 7. The language $S_i' \subseteq S_i$ is *relatively unambiguous* with respect to S_i, alphabets Σ^{s} and $\Sigma^{\mathrm{s}}_{i,\mathrm{uc}} = \Sigma^{\mathrm{s}} \backslash \Sigma^{\mathrm{s}}_{i,\mathrm{c}}$, $\mathcal{L}(\mathbf{M}_i)$ and the projection $P_{i\mathrm{s}} : \Sigma_i^* \to (\Sigma^{\mathrm{s}})^*$ if, for any $s, s' \in \overline{S_i}$ such that $P_{i\mathrm{s}}(s) = P_{i\mathrm{s}}(s')$,

(1) $(\forall \sigma \in \Sigma^{\mathrm{s}})\ s\sigma \in \overline{S_i'} \ \Rightarrow (\exists s'' \in \Sigma_i^*)$
$s' \leq s''$, $P_{i\mathrm{s}}(s'') = P_{i\mathrm{s}}(s')$ and $s''\sigma \in \overline{S_i'}$;
(2) $s \in S_i' \Rightarrow (\exists s'' \in \Sigma_i^*)$
$s'' \leq s'$or $s' < s''$, $P_{i\mathrm{s}}(s'') = P_{i\mathrm{s}}(s')$, and $s'' \in S_i'$;

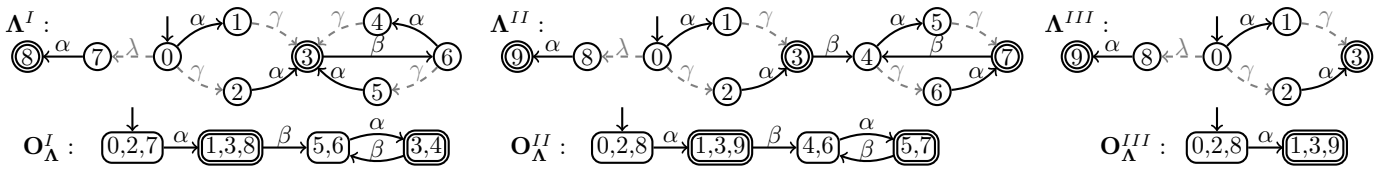

Fig. 6. Illustrative example for Alg. 2 showing $\mathbf{\Lambda}$ (top) and $\mathbf{O}_{\mathbf{\Lambda}}$ (bottom). $\mathbf{\Lambda}^I$ does not satisfy Cond. 1 – state 3 appears in more than one state of $\mathbf{O}_{\mathbf{\Lambda}}$. Then $\mathbf{\Lambda}^{II}$ shows $\mathbf{O}_{\mathbf{\Lambda}} \parallel \mathbf{\Lambda}$, where Cond. 1 holds. $\mathbf{\Lambda}^{III}$ shows EnforceUnambiguity($\mathbf{\Lambda}$).

(3) $(\forall \sigma \in \Sigma^{\mathrm{s}}_{i,\mathrm{uc}})\ \left[s\sigma \in \overline{S'_i} \text{ and } s'\sigma \in \mathcal{L}(\mathbf{M}_i)\right] \Rightarrow s'\sigma \in \overline{S'_i}$.

Algorithm 2 EnforceUnambiguity($\mathbf{\Lambda}$)

Require: Automaton $\mathbf{\Lambda}$, its plant $\mathbf{M}^\ell$, subalphabet $\Sigma^{\mathrm{s}} = \Sigma^{\mathrm{s}}_{\mathrm{c}} \dot\cup \Sigma^{\mathrm{s}}_{\mathrm{uc}}$
1: Compute $\mathbf{O}_{\mathbf{\Lambda}}$
2: **if** $\exists q, q' \in \mathbf{O}_{\mathbf{\Lambda}}.Q$, with $q \neq q'$, such that $q \cap q' \neq \emptyset$ **then**
3: $\quad \mathbf{\Lambda} \leftarrow \mathbf{\Lambda} \,||\, \mathbf{O}_{\mathbf{\Lambda}}$ and recompute $\mathbf{O}_{\mathbf{\Lambda}}$
4: **end if**
5: $M_{\mathbf{O}} \leftarrow \emptyset$
6: **for all** $q \in \mathbf{O}_{\mathbf{\Lambda}}.Q_{\mathrm{m}}$ **do**
7: $\quad q_{\mathrm{m}} \leftarrow \{x \in q \mid x \in \mathbf{\Lambda}.Q_{\mathrm{m}}\}$ and $q_{\mathrm{m}?} \leftarrow q \backslash q_{\mathrm{m}}$
8: $\quad$ **if** AmbigMarking($\mathbf{\Lambda}, q_{\mathrm{m}?}, q_{\mathrm{m}}$) **then**
9: $\quad\quad M_{\mathbf{O}} \leftarrow M_{\mathbf{O}} \cup \{q\}$
10: $\quad$ **end if**
11: **end for**
12: $\mathbf{O}_{\mathbf{\Lambda}}.Q_{\mathrm{m}} \leftarrow \mathbf{O}_{\mathbf{\Lambda}}.Q_{\mathrm{m}} \backslash M_{\mathbf{O}}$ and $\mathbf{O}_{\mathbf{\Lambda}} \leftarrow$ trim($\mathbf{O}_{\mathbf{\Lambda}}$) and $\mathbf{O} \leftarrow \emptyset$
13: $\tilde{\mathbf{\Lambda}} \leftarrow$ plantify($\mathbf{\Lambda}, \mathbf{M}^\ell, \Sigma^{\mathrm{s}}_{\mathrm{uc}}$) and $\tilde{\mathbf{\Lambda}} \leftarrow \tilde{\mathbf{\Lambda}} \parallel \mathbf{O}_{\mathbf{\Lambda}}$
14: **while** $\mathbf{O}_{\mathbf{\Lambda}} \neq \mathbf{O}$ **do**
15: $\quad \mathbf{O} \leftarrow \mathbf{O}_{\mathbf{\Lambda}}$ and $R_{\mathbf{O}} \leftarrow \emptyset$
16: $\quad$ **for all** $q \in \mathbf{O}.Q$ **do**
17: $\quad\quad$ **for all** $\sigma \in \Sigma^{\mathrm{s}}$ such that $\mathbf{O}.\delta(q,\sigma)!$ **do**
18: $\quad\quad\quad q_\sigma \leftarrow \{x \in q \mid \tilde{\mathbf{\Lambda}}.\delta(x,\sigma) \in \mathbf{O}.\delta(q,\sigma)\}$ and $q_{\sigma?} \leftarrow q \backslash q_\sigma$
19: $\quad\quad\quad$ **if** $\sigma \in \Sigma^{\mathrm{s}}_{\mathrm{uc}}$ **then**
20: $\quad\quad\quad\quad q_{\mathrm{u}} \leftarrow \{x \in q \mid \tilde{\mathbf{\Lambda}}.\delta(x,\sigma)! \text{ and } \tilde{\mathbf{\Lambda}}.\delta(x,\sigma) \notin \mathbf{O}.\delta(q,\sigma)\}$
21: $\quad\quad\quad$ **else** $q_{\mathrm{u}} \leftarrow \emptyset$
22: $\quad\quad\quad$ **end if**
23: $\quad\quad\quad$ **if** $q_{\mathrm{u}} \neq \emptyset$ or AmbigPath($\tilde{\mathbf{\Lambda}}, q_{\sigma?}, q_\sigma$) **then**
24: $\quad\quad\quad\quad R_{\mathbf{O}} \leftarrow R_{\mathbf{O}} \cup \left\{\left(q, \sigma, \mathbf{O}.\delta(q,\sigma)\right)\right\}$
25: $\quad\quad\quad$ **end if**
26: $\quad\quad$ **end for**
27: $\quad$ **end for**
28: $\quad \mathbf{O}_{\mathbf{\Lambda}}.\delta \leftarrow \mathbf{O}_{\mathbf{\Lambda}}.\delta \backslash R_{\mathbf{O}}$
29: $\quad \mathbf{O}_{\mathbf{\Lambda}} \leftarrow$ trim($\mathbf{O}_{\mathbf{\Lambda}}$) and $\tilde{\mathbf{\Lambda}} \leftarrow \tilde{\mathbf{\Lambda}} \parallel \mathbf{O}_{\mathbf{\Lambda}}$
30: **end while**
31: **if** $\tilde{\mathbf{\Lambda}} \neq \mathbf{\Lambda}$ **then return** $\tilde{\mathbf{\Lambda}}$ and `True`
32: **else return** $\tilde{\mathbf{\Lambda}}$ and `False`
33: **end if**

In order to enforce relative unambiguity – hence, also enforcing unambiguity and the observability condition from Lem. 1 – we define the function EnforceUnambiguity, in Alg. 2, which manipulates a given automaton $\mathbf{\Lambda}$ into an automaton $\tilde{\mathbf{\Lambda}}$ such that $\tilde{\Lambda}$ is the supremal relatively unambiguous sublanguage of Λ, denoted by $\tilde{\Lambda} = \sup\mathcal{U}(\Lambda)$.

To start explaining Alg. 2, first consider an automaton $\mathbf{\Lambda}$ and a natural projection $P : \Sigma^* \to \Sigma^{\mathrm{s}}$. Let us define the *uncertainty set* of states that a string s in $\mathcal{L}(\mathbf{\Lambda})$ can reach (Cai et al. (2015)) by $U(s) := \left\{\delta(q_0, s') \mid \exists s' \in \mathcal{L}(\mathbf{\Lambda}) : P(s') = P(s)\right\} \subseteq Q$. Now, take as $\mathbf{\Lambda}$ the example in Fig. 7, where only the event a is not in the shared alphabet Σ^{s}. The automaton $\mathbf{\Lambda}$ is ambiguous, because the strings $\lambda\zeta$ and $a\lambda\zeta$ look the same under the projection P, but the former can be followed by θ, while the latter cannot. To obtain the automaton that accepts $\sup\mathcal{U}(\mathbf{\Lambda})$, the event θ must be removed after the string $\lambda\zeta$. However, if we do that in the automaton $\mathbf{\Lambda}$, the event θ would be unnecessarily removed after the string $\gamma\alpha$ – and therefore would have to be removed after $a\gamma\alpha$ as well, overly restricting the behaviour. In order to obtain the supremal sublanguage, we need to adopt for the automaton at hand the same condition as used in Cai et al. (2015); Takai and Ushio (2003).

Cond. 1. $\left(\forall s, t \in \mathcal{L}(\mathbf{\Lambda})\right)\ \delta(q_0, s) = \delta(q_0, t) \Rightarrow U(s) = U(t)$

If this is not satisfied by $\mathbf{\Lambda}$, as in our example, it can be imposed with no loss of generality – but at a high computational cost – by replacing $\mathbf{\Lambda}$ with $\mathbf{O}_{\mathbf{\Lambda}} \parallel \mathbf{\Lambda}$ – see Takai and Ushio (2003) for the proof. We guarantee Cond. 1 holds in the if-loop starting in line 2 of Alg. 2.

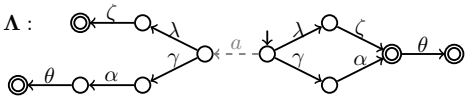

Fig. 7. Example illustrating why Cond. 1 is needed.

Besides being used for Cond. 1, the observer $\mathbf{O}_{\mathbf{\Lambda}}$ also informs which states in $\mathbf{\Lambda}$ are reached from the initial state by strings with the same projection over the shared alphabet, which is needed to check relative unambiguity. The first condition of this property that is enforced is (2). In the loop starting in line 6, we check each marked state q in $\mathbf{O}_{\mathbf{\Lambda}}$. If q contains a nonmarked state from $\mathbf{\Lambda}$ that does not satisfy that condition, q and all the states it contains are respectively unmarked in $\mathbf{O}_{\mathbf{\Lambda}}$ and, through the synchronous product, in $\mathbf{\Lambda}$ (on lines 12 and 13). For such check over the states of q, we call the function AmbigMarking, which performs backward and forward searches to detect if for each state in $q_{\mathrm{m}?}$ there is a sequence of nonshared events that connects this state to any state in q_{m}, either from the former to the latter, or the other way round; if there is *no* such sequence, then and only then the function returns true, i.e., the condition is not satisfied.

Now, note that we intend $\mathbf{S}_i$ to be the input $\mathbf{\Lambda}$ for this algorithm, where $\mathbf{S}_i$ is the output from Negotiation, which is computed assuming all shared events are controllable. Therefore, to check (3) from Def. 7 we need to remember which transitions by events in $\Sigma^{\mathrm{s}}_{i,\mathrm{uc}}$ that are possible in the local plant $\mathbf{M}_i$ were removed from $\mathbf{S}_i$. This is done by the plantify function in line 13. This function adds a blocking state to a copy of $\mathbf{\Lambda}$ denoted by $\tilde{\mathbf{\Lambda}}$ and, for every state where an event in $\Sigma^{\mathrm{s}}_{\mathrm{uc}}$ (which is $\Sigma^{\mathrm{s}}_{i,\mathrm{uc}}$ for $\mathbf{S}_i$) is not enabled, adds a transition by such event from this state to the blocking one; finally, it composes the modified automaton with $\mathbf{M}^\ell$ (i.e., $\mathbf{M}_i$ for $\mathbf{S}_i$), so the only newly added transitions left are the ones allowed in the local plant. Notice that when the automaton $\tilde{\mathbf{\Lambda}}$ resulting from this function is composed with $\mathbf{O}_{\mathbf{\Lambda}}$, only the newly added transitions by shared events that are not relevant for the

condition (3) from Def. 7 are removed – because if they are not in $\mathbf{O_\Lambda}$, the premise denoted by "$s\sigma \in \overline{S_i'}$" in that condition is false, and the implication is satisfied; to avoid confusion, note that here the term "S_i'" from the definition refers to $\tilde{\Lambda}$.

To check if conditions (1) and (3) from Def. 7 hold, in the loop starting in line 16 we inspect each state q in $\mathbf{O_\Lambda}$, which is a set of states in $\tilde{\mathbf{\Lambda}}$. If a shared event σ is enabled from q, there is at least one state in q from which a transition with σ is defined in $\tilde{\mathbf{\Lambda}}$. We can thus partition q into a set q_σ of states in $\tilde{\mathbf{\Lambda}}$ from which σ is enabled, and a set $q_{\sigma?}$ with the remaining states. Then (1) is not disrespected if from every state in $q_{\sigma?}$ it is possible to reach (by a sequence of nonshared events) some state in q_σ; this is checked by the function AmbigPath through a backward search from states in q_σ – the function returns true if and only if such reach is *not* possible. If (1) is disrespected, the transitions with σ from q and from the states in q_σ need to be removed from $\mathbf{O_\Lambda}$ and $\tilde{\mathbf{\Lambda}}$, respectively, which is done in lines 28 and 29. In order to impose condition (3), such transitions also need to be removed if σ is an uncontrollable event in $\Sigma_{\text{uc}}^{\text{s}}$ and if for some state in $q_{\sigma?}$ this event is enabled, which means it leads to the blocking state added by the plantify function.

As an illustrative example for this algorithm, see Fig. 6. It is worth to notice that an automaton resulting from this algorithm may not be trim, for we only eliminate from $\tilde{\mathbf{\Lambda}}$ transitions by shared events – that disrespect relative unambiguity – by taking its composition with $\mathbf{O_\Lambda}$. There are two reasons for that. Firstly, transitions by local events can only be removed if we take into account the controllability issues it may bring; thus, this trimming problem is left for the Negotiation function. Secondly, although it is correct to remove states from the observer to enforce the property in question, we cannot remove states from $\tilde{\mathbf{\Lambda}}$ contained in a state of $\mathbf{O_\Lambda}$ that survived the trimming; such states are needed to remember the original sequences generated by $\mathbf{\Lambda}$ – that form the language denoted by $\overline{S_i}$ in Def. 7 – in order to compare them to the sequences generated by the iteratively refined automaton $\tilde{\mathbf{\Lambda}}$ – that form the language denoted by $\overline{S_i'}$ in Def. 7. As a last remark, note that all the transitions added by the function plantify are removed from $\tilde{\mathbf{\Lambda}}$ before it is returned by Alg. 2: either they are not possible in the respective states in $\mathbf{O_\Lambda}$ – which is computed over $\mathbf{\Lambda}$, where these transitions are not present – or they are, which means they disrespect (3) from Def. 7 and, therefore, are finally removed.

5.4 Negotiation of Controllable & Unambiguous Supervisors

Now that we can enforce the relative unambiguity property to guarantee nonconflict and cooperation, we can combine the functions Negotiation and EnforceUnambiguity to compute local supervisors which ensure a nonblocking global closed-loop behaviour. This might, though, sacrifice maximal permissiveness in order to prevent blocking situations in a purely decentralised manner. The final algorithm, given in Alg. 3, is denoted by CBSS as an initialism for *Contract-Based Supervisor Synthesis* [3].

Algorithm 3 CBSS

Require: Automata $\mathbf{K}_1$ and $\mathbf{K}_2$
1: $(\mathbf{S}_1', \mathbf{S}_2') \leftarrow$ Negociate$(\mathbf{K}_1, \mathbf{K}_2)$
2: $(\mathbf{S}_1, cond1) \leftarrow$ EnforceUnambiguity$(\mathbf{S}_1')$
3: $(\mathbf{S}_2, cond2) \leftarrow$ EnforceUnambiguity$(\mathbf{S}_2')$
4: **while** *cond*1 or *cond*2 **do**
5: $(\mathbf{S}_1', \mathbf{S}_2') \leftarrow$ Negociate$(\mathbf{S}_1, \mathbf{S}_2)$
6: **for** $i \in \{1, 2\}$ **do**
7: **if** $\mathbf{S}_i \not\equiv \mathbf{S}_i'$ **then**
8: $(\mathbf{S}_i, condi) \leftarrow$ EnforceUnambiguity$(\mathbf{S}_i')$
9: **end if**
10: **end for**
11: **end while**
12: **return** $\mathbf{S}_1$ and $\mathbf{S}_2$

Combining multiple previous results, we have the following soundness result of Alg. 3. It is the main implication of this section, for it shows that the fully local supervisors impose a globally nonblocking closed-loop behaviour that respects the local specifications.

Theorem 3. Consider a plant composed of two subsystems $\mathbf{M}_i$ with plantified specifications $\mathbf{K}_i$, as in Sec. 3.1. Let $(\mathbf{S}_1, \mathbf{S}_2) = \text{CBSS}(\mathbf{K}_1, \mathbf{K}_2)$. Further, let f_i be the local supervisors defined from $\mathbf{S}_i$ via Def. 5. Then, it holds that

(1) $\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2) \in \mathcal{C}(K_1 \parallel K_2)$, and
(2) $\mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2) = \overline{\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2)}$.

6. EXPERIMENTAL RESULTS

We have implemented our negotiation framework in `Supremica` (Malik et al. (2017)) and run it on a modified version of the classical *cat & mouse* example from Moody and Antsaklis (1998). Here, an equal number of cats and mice move in a building with multiple rooms connected by doors and passages, and the goal is to ensure that cat and mouse are never simultaneously in the same room.

The building is designed as follows. For an integer $n \geq 0$, it consists of $1+2n$ blocks, which are divided into 5 rooms and identified by a number $k \in [-n, n]$, with $k = 0$ being the central block. If the building has more than one block, i.e., if $n > 0$, then for all room $r \in [1, 4]$, and for all block $k \in [-n, n)$ such that $k \equiv 4 - r \ (mod\ 4)$, there is a bidirectional passage both for cats and mice connecting

[3] Note that, in Alg. 3, each boolean variables *condi* is set to `False` only if the function EnforceUnambiguity does not change the corresponding output of Negotiation, in this case denoted by $\mathbf{S}_i'$. However, if *condi* = `False` and *condj* = `True` for $i \neq j$, Negotiation is called again and perform $\text{CSynth}(\mathbf{S}_i') = \mathbf{S}_i'$ before entering the main loop, which is a wasteful execution. To avoid this, we can simply add optional arguments to the function Negotiation and replace line 1 in Alg. 1 by the preamble below; then, in line 5 from Alg. 3, we call $\text{Negotiation}(S_1, S_2, (cond1, cond2))$.

Optional: Pair of boolean arguments $(opt1, opt2)$; default value $(true, true)$
if *opt*1 **then** $\mathbf{S}_1 \leftarrow \text{CSynth}(\mathbf{S}_1^{\text{init}})$
else $\mathbf{S}_1 \leftarrow \mathbf{S}_1^{\text{init}}$
end if
if *opt*2 **then** $\mathbf{S}_2 \leftarrow \text{CSynth}(\mathbf{S}_2^{\text{init}})$
else $\mathbf{S}_2 \leftarrow \mathbf{S}_2^{\text{init}}$
end if

the rooms r from the blocks k and $k+1$. As depicted in Fig. 8, a passage or door may allow either cats, mice, or both to go from a room to the other; the arrows indicate the possible direction for each type of animal — dashed arrows for mice and solid ones for cats. The events in our model are all possible combinations of a passage or a door, an animal – not just a type of animal – and a direction allowed by that passage or door. For any block, the events related to the door between rooms 1 and 3 are (always local and) uncontrollable, while all the other events in the plant are controllable by some subsystem. The initial and the marked state are characterised by all the cats being in room 2 of block n and all the mice in room 4 of block $-n$, while all the other rooms are empty.

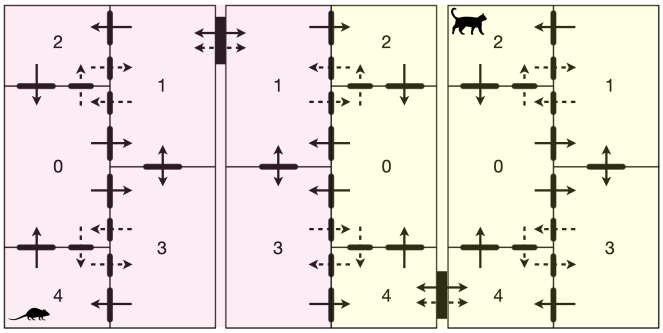

Fig. 8. Building design used for the *cat & mouse* example.

Due to the restriction of only two subsystems in this work, we divide the given building into two parts, so that only the events associated with the doors that connect rooms from different subsystems are observable by both, making them shared events. We see the plant as a ground-floor (horizontal) building, where the blocks lie next to one another – for an even k, the block k is identical to the central one, and for an odd k, the block is mirrored. Then, the first subsystem is composed by all the blocks with $k < 0$ and by the rooms $r \in \{1, 3\}$ of block 0 – indicated by pink colour in Fig. 8 – while the second one is composed by the blocks with $k > 0$ and by the rooms $r \in \{0, 2, 4\}$ in the central block – indicated by yellow. The *shared* doors that allow mice are controllable only by the first subsystem, whilst the *shared* doors that allow cats, only by the second, so that both of them control the same number of shared events. The contract we compute for this example models the opening and closing behaviour of doors connecting the two parts of the building.

Taking the scenario of a single cat and mouse in a single block building as reference, we varied the example such that we increase the total number of events in the system ($\Sigma = \Sigma_1 \cup \Sigma_2$) but, in Case A (single-block plant, multiple cats and mice), we keep the number of shared events proportional, and, in Case B (3-block plant, single cat and mouse), this number remains fixed. The results are reported in Table 1, where $|A|$ denotes the size of a set A, and $\mathbf{\Lambda}.Q$ the state set of an automaton $\mathbf{\Lambda}$; the last row is the execution time in milliseconds. All experiments were run on a 2 GHz Quad-Core Intel i5.

Our experiments should be viewed as a proof-of-concept. We are computing local supervisors monolithically per subsystem, which obviously does not scale when subsystem sizes increase. Improvements that use existing scalable approaches – such as Mohajerani et al. (2014), for example

Table 1. Cat & Mouse experimental results

	Case A				Case B
# cats	1	2	5	10	1
$\lvert\Sigma^s\rvert/\lvert\Sigma\rvert$	8/22	16/44	40/110	80/220	8/58
$\lvert\mathbf{M}_1.Q\rvert$	16	36	144	484	10240
$\lvert\mathbf{M}_2.Q\rvert$	56	189	1512	9317	39256
$\lvert\mathbf{C}_1^0.Q\rvert$	26	108	1073	8870	8464
$\lvert\mathbf{C}_2^0.Q\rvert$	6	13	46	141	126
$\lvert\mathbf{C}_1.Q\rvert = \lvert\mathbf{C}_2.Q\rvert$	4	8	26	76	17
$\lvert\mathbf{S}_1.Q\rvert$	5	11	41	131	85
$\lvert\mathbf{S}_2.Q\rvert$	5	11	41	131	74
Time (ms)	33	100	1133	78548	10100

– along with additional dynamic algorithms that reuse computations from previous iterations of the negotiation algorithm are currently under development. Nevertheless, in these preliminary results we already see that the ratio of shared to nonshared events largely impacts the computation time when looking at instances of Case A and Case B with similar subsystem sizes. A larger number of shared events leads to larger contracts. It is therefore expected that our negotiation-based decentralised synthesis approach works best if the number of shared events is comparably low.

REFERENCES

Apaza-Perez, W.A., Combastel, C., and Zolghadri, A. (2020). On distributed symbolic control of interconnected systems under persistency specifications. *International Journal of Applied Mathematics and Computer Science*, 30(4).

Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T.A., Larsen, K.G., et al. (2018). Contracts for system design. *Foundations and Trends in Electronic Design Automation*, 12(2-3), 124–400.

Cai, K. and Wonham, W.M. (2016). *Supervisor Localization, A Top-Down Approach to Distributed Control of Discrete-Event Systems*. Lecture Notes in Control and Information Sciences. Springer.

Cai, K., Zhang, R., and Wonham, W.M. (2015). Relative observability of discrete-event systems and its supremal sublanguages. *IEEE Transactions on Automatic Control*, 60(3), 659–670.

Cassandras, C.G. and Lafortune, S. (2021). *Introduction to Discrete Event Systems*. Springer, third edition.

Finkbeiner, B. and Passing, N. (2021). Compositional synthesis of modular systems. In *International Symposium on Automated Technology for Verification and Analysis*, 303–319. Springer.

Komenda, J. and Masopust, T. (2017). Computation of controllable and coobservable sublanguages in decentralized supervisory control via communication. *Discrete Event Dynamic Systems*, 27.

Majumdar, R., Mallik, K., Schmuck, A.K., and Zufferey, D. (2020). Assume-guarantee distributed synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11), 3215–3226.

Malik, R., Akesson, K., Flordal, H., and Fabian, M. (2017). Supremica–an efficient tool for large-scale discrete event systems. *IFAC-PapersOnLine*, 50(1), 5794–5799. 20th IFAC World Congress.

Mohajerani, S., Malik, R., and Fabian, M. (2014). A framework for compositional synthesis of modular nonblocking supervisors. *Automatic Control, IEEE Transactions on*, 59, 150–162.

Moody, J.O. and Antsaklis, P.J. (1998). *Example Applications*, 113–152. Springer US, Boston, MA.

Queiroz, M.H. and Cury, J.E.R. (2000). Modular supervisory control of large scale discrete-event systems. In *Proc. 5th International Workshop on Discrete Event Systems (WODES'2000)*, 103–110. Ghent, Belgium.

Ramadge, P.J. and Wonham, W.M. (1987). Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25, 206–230.

Sangiovanni-Vincentelli, A., Damm, W., and Passerone, R. (2012). Taming dr. frankenstein: Contract-based design for cyber-physical systems. *European journal of control*, 18(3), 217–238.

Schmidt, K., Moor, T., and Perk, S. (2008). Nonblocking hierarchical control of decentralized discrete event systems. *IEEE Transactions on Automatic Control*, 53(10), 2252–2265.

Su, R. and Thistle, J. (2006). A distributed supervisor synthesis approach based on weak bisimulation. In *2006 8th International Workshop on Discrete Event Systems*, 64–69.

Takai, S. and Ushio, T. (2003). Effective computation of an lm(g)-closed, controllable, and observable sublanguage arising in supervisory control. *Systems and Control Letters - SYST CONTROL LETT*, 49, 191–200.

Wonham, W.M. and Cai, K. (2019). *Supervisory control of discrete-event systems*. Communications and Control Engineering. Springer.

Wonham, W. and Ramadge, P. (1988). Modular supervisory control of discrete-event systems. *Math. Control Signal Systems*, 1, 13–30.

Appendix A. PROOFS

Remark 1. Let $\mathbf{S}_i$ be the automata defined on Alg. 1 at any stage of its execution given the inputs $\mathbf{K}_i$, which are the plantified specifications for the subsystems $\mathbf{M}_i$. Then $S_i \subseteq K_i \subseteq M_i$.

Proof. As explained in Sec. 2, each plantified specification $\mathbf{K}_i$ is the composition of $\mathbf{M}_i$ and all the local specifications $\mathbf{E}_i$, so clearly $K_i \subseteq M_i$. NEGOTIATION only performs, iteratively, composition and CSYNTH operations, which only remove strings in the semantics of its inputs $\mathbf{K}_i$, so the semantics of $\mathbf{S}_i$ are subsets of the former languages. Thus, we have that $S_i \subseteq K_i$. □

Remark 2. (Result used in the proof of Theorem 1.) Given languages A, B and $C = \overline{C}$ over $\Sigma = \Sigma_c \dot\cup \Sigma_{uc}$ such that $A, B \subseteq C$, we have that

$$\sup\mathcal{C}(A) \subseteq B \;\Leftrightarrow\; \sup\mathcal{C}(A) \subseteq \sup\mathcal{C}(B)\,,$$

where controllability is taken with respect to Σ_{uc} and C.

Proof.
($\Rightarrow$) $\sup\mathcal{C}(A)$ is a sublanguage of B by the premiss; besides, it is controllable, so $\sup\mathcal{C}(A) \in \mathcal{C}(B)$. Since controllability is closed with respect to union, $\sup\mathcal{C}(B) = \bigcup_{L\in\mathcal{C}(B)} L \supseteq \sup\mathcal{C}(A)$.

($\Leftarrow$) Assume $\sup\mathcal{C}(A) \subseteq \sup\mathcal{C}(B)$. Then, $\sup\mathcal{C}(A) \in \mathcal{C}(B)$, so $\sup\mathcal{C}(A) \subseteq B$. □

Remark 3. (Result used in the proof of Theorem 1.) In the context of the plant defined in Sec. 3.1, and for any languages Λ_i over Σ_i such that $\Lambda_i \subseteq K_i$, $i \in \{1, 2\}$, we have that

$$P_i\big(\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)\big) \subseteq \sup\mathcal{C}_{\mathbf{M}_i,\Sigma^\ell_{i,uc}}(\Lambda_i),$$

where we denote by P_i the projection $P_i : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_i^*$, the global plant by $\mathbf{M} = \mathbf{M}_1 \parallel \mathbf{M}_2$, with uncontrollable events $\Sigma_{uc} = \Sigma^\ell_{1,uc} \dot\cup \Sigma^\ell_{2,uc}$, and where $\sup\mathcal{C}_{\Lambda,\Sigma_u}$ indicates that controllability is taken with respect to a language Λ and and uncontrollable alphabet Σ_u.

Proof.
(i) $\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2) \subseteq \Lambda_1 \parallel \Lambda_2$, so $\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2) \subseteq P_i^{-1}(\Lambda_i) \Rightarrow P_i\big(\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)\big) \subseteq \Lambda_i$.

(ii) By the definition of $\sup\mathcal{C}$ and controllability, we have that, for all $u \in \overline{\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)}$ and $\sigma \in \Sigma_{uc}$, if $u\sigma \in \mathcal{L}(\mathbf{M})$, then $u\sigma \in \overline{\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)}$. Besides, $P_i(u) \in P_i\big(\overline{\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)}\big) = \overline{P_i\big(\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)\big)}$, and $u \in \mathcal{L}(\mathbf{M})$, because $\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2) \subseteq K_1 \parallel K_2 \subseteq \mathcal{L}(\mathbf{M})$, which is prefix-closed by definition. Note that $\sigma \in \Sigma^\ell_{i,uc}$ for some i, that is, σ is a local event and hence $\sigma \notin \Sigma_j$ for $j \neq i$; then $P_i(u)\sigma \in \mathcal{L}(\mathbf{M}_i) \Leftrightarrow u\sigma \in \mathcal{L}(\mathbf{M})$. Now, by the definition of $\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)$, we have that $P_i(u)\sigma \in \mathcal{L}(\mathbf{M}_i) \Leftrightarrow u\sigma \in \mathcal{L}(\mathbf{M}) \Rightarrow u\sigma \in \overline{\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)} \Rightarrow P_i(u)\sigma \in P_i\big(\overline{\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)}\big) = \overline{P_i\big(\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)\big)}$, implying $P_i\big(\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2\big)$ is controllable with respect to $\mathcal{L}(\mathbf{M}_i)$ and $\Sigma^\ell_{i,uc}$.

From (i) and (ii), we have that $P_i\big(\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(\Lambda_1 \parallel \Lambda_2)\big) \subseteq \sup\mathcal{C}_{\mathbf{M}_i,\Sigma^\ell_{i,uc}}(\Lambda_i)$. □

Theorem 1. Consider a plant composed of two subsystems $\mathbf{M}_i$ and their corresponding plantified specifications $\mathbf{K}_i$, as described in Sec. 3.1. Let $\mathbf{S}_i$ be the outputs of Algorithm 1 for the inputs $\mathbf{K}_i$, namely, $(\mathbf{S}_1, \mathbf{S}_2) = \text{NEGOTIATION}(\mathbf{K}_1, \mathbf{K}_2)$. Then, we have that

$$\sup\mathcal{C}(S_1 \parallel S_2) = \sup\mathcal{C}(K_1 \parallel K_2),$$

where $\sup\mathcal{C}$ is taken with respect to $\mathcal{L}(\mathbf{M}_1 \parallel \mathbf{M}_2)$ and $\Sigma^\ell_{1,uc} \cup \Sigma^\ell_{2,uc}$.

Proof.
($\Rightarrow$) From Remark 1, it is straightforward to see that $\sup\mathcal{C}(S_1 \parallel S_2) \subseteq \sup\mathcal{C}(K_1 \parallel K_2)$, as $S_1 \parallel S_2 \subseteq K_1 \parallel K_2$.

($\Leftarrow$) Let us now show that $\sup\mathcal{C}(K_1 \parallel K_2) \subseteq \sup\mathcal{C}(S_1 \parallel S_2)$. The proof is by induction over the steps (rounds) of the Alg. 1. We denote by S_i^k the language accepted by the automaton $\mathbf{S}_i$ from NEGOTIATION at the end of each round $k \geq 1$, which we consider to be at line 10. Moreover, we use the notation $\mathbf{M}$, Σ_{uc}, and $\sup\mathcal{C}_{\Lambda,\Sigma_u}$ as in Remark 3.

Step 1:

$$\begin{aligned} S_1^1 \parallel S_2^1 &= \sup\mathcal{C}_{\mathbf{M}_1,\Sigma^\ell_{1,uc}}(K_1) \parallel P_{2s}(\sup\mathcal{C}_{\mathbf{M}_2,\Sigma^\ell_{2,uc}}(K_2)) \parallel \\ &\qquad \sup\mathcal{C}_{\mathbf{M}_2,\Sigma^\ell_{2,uc}}(K_2) \parallel P_{1s}(\sup\mathcal{C}_{\mathbf{M}_1,\Sigma^\ell_{1,uc}}(K_1)) \\ &= \sup\mathcal{C}_{\mathbf{M}_1,\Sigma^\ell_{1,uc}}(K_1) \parallel \sup\mathcal{C}_{\mathbf{M}_2,\Sigma^\ell_{2,uc}}(K_2) \\ &\supseteq \sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(K_1 \parallel K_2)\,. \qquad \text{(by Remark 3)} \end{aligned}$$

Hence, from Remark 2 we have that $\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(K_1 \parallel K_2) \subseteq \sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(S_1^1 \parallel S_2^1)$.

Step k: assume $\sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(K_1 \parallel K_2) \subseteq \sup\mathcal{C}_{\mathbf{M},\Sigma_{uc}}(S_1^k \parallel S_2^k)$.

Step $k+1$:

$$
\begin{aligned}
S_1^{k+1} \parallel S_2^{k+1} &= \sup\mathcal{C}_{\mathbf{M}_1,\Sigma^\ell_{1,\mathrm{uc}}}(S_1^k) \parallel P_{2\mathrm{s}}(\sup\mathcal{C}_{\mathbf{M}_2,\Sigma^\ell_{2,\mathrm{uc}}}(S_2^k)) \parallel \\
&\qquad \sup\mathcal{C}_{\mathbf{M}_2,\Sigma^\ell_{2,\mathrm{uc}}}(S_2^k) \parallel P_{1\mathrm{s}}(\sup\mathcal{C}_{\mathbf{M}_1,\Sigma^\ell_{1,\mathrm{uc}}}(S_1^k)) \\
&= \sup\mathcal{C}_{\mathbf{M}_1,\Sigma^\ell_{1,\mathrm{uc}}}(S_1^k) \parallel \sup\mathcal{C}_{\mathbf{M}_2,\Sigma^\ell_{2,\mathrm{uc}}}(S_2^k) \\
&\supseteq \sup\mathcal{C}_{\mathbf{M},\Sigma_{\mathrm{uc}}}(S_1^k \parallel S_2^k) \qquad \text{(by Remark 3)} \\
&\supseteq \sup\mathcal{C}_{\mathbf{M},\Sigma_{\mathrm{uc}}}(K_1 \parallel K_2)\,.
\end{aligned}
$$

So, $\sup\mathcal{C}_{\mathbf{M},\Sigma_{\mathrm{uc}}}(K_1 \parallel K_2) \subseteq \sup\mathcal{C}_{\mathbf{M},\Sigma_{\mathrm{uc}}}(S_1^{k+1} \parallel S_2^{k+1})$. □

Lemma 1. In the context of Definition 5, the following equalities hold if and only if S_i are observable with respect to $\mathcal{L}(\mathbf{M}_i)$, $\Sigma^{\mathrm{s}}\backslash\Sigma^{\mathrm{s}}_{i,\mathrm{c}}$ and $P_{i\mathrm{s}} : \Sigma_i^* \to (\Sigma^{\mathrm{s}})^*$:

(1) $\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2) = S_1 \parallel S_2$ and
(2) $\mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2) = \overline{S_1} \parallel \overline{S_2}$.

Proof. For this proof, we define the projections $P_i : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_i^*$ (as in Remark 3), and $P_{\mathrm{s}} : (\Sigma_1 \cup \Sigma_2)^* \to (\Sigma^{\mathrm{s}})^*$.

($\Rightarrow$) Let us first show that the observability property at hand implies that (1) and (2) are valid.

(a) $\overline{S_1} \parallel \overline{S_2} \subseteq \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$:

If there exists $i \in \{1,2\}$ such that $S_i = \emptyset$, then $\overline{S_1} \parallel \overline{S_2} = \emptyset$ and it trivially holds. Consider then $\overline{S_i} \neq \emptyset$ for all $i \in \{1,2\}$. Then, $\epsilon \in \overline{S_1} \parallel \overline{S_2}$, but $\epsilon \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$ by definition of $\mathcal{L}(f_i/\mathbf{M}_i)$.

Let us say $\overline{S_1} \parallel \overline{S_2} \supset \{\epsilon\}$, and let us prove that for any $s \in \overline{S_1} \parallel \overline{S_2}$ such that $s \neq \epsilon$, $s \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$. This proof is by induction over the lenght of $s \neq \epsilon$. Note that $\overline{S_i} \subseteq \mathcal{L}(\mathbf{M}_i)$, so $s \in \mathcal{L}(\mathbf{M}_1) \parallel \mathcal{L}(\mathbf{M}_2)$, i.e., $P_i(s) \in \mathcal{L}(\mathbf{M}_i)$ for all i.

Base case. If $s = \alpha \in \Sigma_i^\ell$ for some i, then $P_i(s) = \alpha = \epsilon\alpha \in \overline{S_i}$, so $\alpha \in f_i(\epsilon)$. Besides, $\epsilon\alpha \in \mathcal{L}(\mathbf{M}_i)$. Since $\epsilon \in \mathcal{L}(f_i/\mathbf{M}_i)$, $\epsilon\alpha = s \in \mathcal{L}(\mathbf{M}_i)$, and $\alpha \in f_i(\epsilon)$, we have that $\epsilon\alpha \in \mathcal{L}(f_i/\mathbf{M}_i)$ by definition of this language. Now, because α is a local event and by definition of synchronous composition, $\epsilon\alpha \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$. On the other hand, if $s = \alpha \in \Sigma^{\mathrm{s}}$, then for all i we have that $P_i(s) = \alpha = \epsilon\alpha \in \overline{S_i}$, so $\alpha \in f_i(\epsilon)$. By the same arguments as above, $\epsilon\alpha \in \mathcal{L}(f_i/\mathbf{M}_i)$ for all i, so $s \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$.

Now, suppose $s \in \Sigma^*\backslash(\Sigma\cup\{\epsilon\})$. We can write $s = s'\alpha$, with $s' \neq \epsilon$, $s' \in \overline{S_1} \parallel \overline{S_2}$. Assume $s' \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$ (inductive hypothesis), then $P_i(s') \in \mathcal{L}(f_i/\mathbf{M}_i)$ for all i. Besides, $P_i(s) \in \overline{S_i}$ for all i, so $\alpha \in f_i(P_i(s'))$ for all i if $\alpha \in \Sigma^{\mathrm{s}}$, or for some i if $\alpha \in \Sigma_i^\ell$. Thus, as in the base case, we can argue that $s \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$.

(b) $\mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2) \subseteq \overline{S_1} \parallel \overline{S_2}$:

By definition, $\epsilon \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$, and since S_i are the marked languages of nonempty automata, $\epsilon \in \overline{S_1} \parallel \overline{S_2}$.

Consider the case where $\mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2) \supset \{\epsilon\}$; let us prove, again by induction, that for any $s \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$ with $s \neq \epsilon$, $s \in \overline{S_1} \parallel \overline{S_2}$.

Base case: $s = \alpha \in \Sigma$. If $\alpha \in \Sigma_i^\ell$ for some i, then $P_i(s) = \alpha \in \mathcal{L}(f_i/\mathbf{M}_i)$, so $\alpha \in f_i(\epsilon)$. Since $\alpha \notin \Sigma^{\mathrm{s}}$, by definition of f_i, $P_i(s) = \alpha = \epsilon\alpha \in \overline{S_i}$, therefore $s \in P_i^{-1}(\overline{S_i})$. Besides, $P_j(s) = \epsilon$ for $j \neq i$, so $s \in P_j^{-1}(\overline{S_j})$. Thus, $s \in \overline{S_1} \parallel \overline{S_2}$. If $\alpha \in \Sigma^{\mathrm{s}}$, then for all i we have that $P_i(s) = \alpha \in \mathcal{L}(f_i/\mathbf{M}_i)$. Moreover, if for all i it is true that $\alpha \in \Sigma_{i,\mathrm{c}}$, then by definition of f_i we have that $P_i(s) = \epsilon\alpha \in \overline{S_i}$, so $s \in \overline{S_1} \parallel \overline{S_2}$. Else, there exists i such that $\alpha \in \Sigma^{\mathrm{s}}\backslash\Sigma_{i,\mathrm{c}}$. Then, $\alpha \in \Sigma^{\mathrm{s}}_{j,\mathrm{c}}$ for $j \neq i$, so by the same argument as above $P_j(s) \in \overline{S_j}$. Since $P_{1\mathrm{s}}(S_1) = P_{2\mathrm{s}}(S_2) \Rightarrow P_{1\mathrm{s}}(\overline{S_1}) = P_{1\mathrm{s}}(\overline{S_2})$, there is a string $\hat{s} \in \Sigma^*$ such that $P_{\mathrm{s}}(\hat{s}) = P_{\mathrm{s}}(s) = \alpha$ and $P_i(\hat{s}) \in \overline{S_i}$. If $\hat{s} = s$, $P_i(s) \in \overline{S_i}$ so $s \in \overline{S_1} \parallel \overline{S_2}$. Else, we can take $u \in \Sigma^*$ such that $u\alpha \leq \hat{s}$ and $P_{\mathrm{s}}(u\alpha) = P_{\mathrm{s}}(\hat{s})$ (so $P_{\mathrm{s}}(u) = \epsilon$). Thus, $P_i(u) \in \overline{S_i}$ and $P_i(u\alpha) = P_i(u)\alpha \in \overline{S_i}$. Moreover, $\epsilon\alpha = P_i(s) \in \mathcal{L}(\mathbf{M}_i)$ (because $P_i(s) \in \mathcal{L}(f_i/\mathbf{M}_i) \subseteq \mathcal{L}(\mathbf{M}_i)$), $\epsilon \in \overline{S_i}$, $\alpha \in \Sigma_{i,\mathrm{uc}}$, and S_i is observable with respect to uncontrollable events $\Sigma^{\mathrm{s}}\backslash\Sigma_{i,\mathrm{c}}$ and $\mathcal{L}(\mathbf{M}_i)$. Therefore, $P_i(s) = \alpha = \epsilon\alpha \in \overline{S_i}$ and hence $s \in \overline{S_1} \parallel \overline{S_2}$.

Now, let us prove for any s such that $s = s'\alpha$ and $s \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$, with $s' \in \Sigma^*\backslash\{\epsilon\}$ and $\alpha \in \Sigma$, that $s \in \overline{S_1} \parallel \overline{S_2}$, assuming $s' \in \overline{S_1} \parallel \overline{S_2}$ (inductive hypothesis). Noting that $P_i(s) = P_i(s')P_i(\alpha)$, we can apply the same arguments (and follow the same scenarios for the event α) as we did in the base case. The single difference is that $P_i(s') \neq \epsilon$, but thanks to the inductive hypothesis, we have that $P_i(s') \in \overline{S_i}$ for all i, so the same conclusions hold. This implies $s \in \overline{S_1} \parallel \overline{S_2}$.

(c) $S_1 \parallel S_2 \subseteq \mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2)$:

For any $s \in S_1 \parallel S_2$, $P_i(s) \in S_i \subseteq \mathcal{L}_m(\mathbf{M}_i)$ for all i. Moreover, because $S_1 \parallel S_2 \subseteq \overline{S_1} \parallel \overline{S_2} = \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$, we have that $s \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$, i.e., $P_i(s) \in \mathcal{L}(f_i/\mathbf{M}_i)$. Thus, $P_i(s) \in \mathcal{L}_m(f_i/\mathbf{M}_i)$ by the definition of this language, so $s \in \mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2)$.

(d) $\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2) \subseteq S_1 \parallel S_2$:

For any $s \in \mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2)$, $P_i(s) \in \mathcal{L}_m(f_i/\mathbf{M}_i) \subseteq \mathcal{L}_m(\mathbf{M}_i)$ for all i. Since $\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2) \subseteq \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2) = \overline{S_1} \parallel \overline{S_2}$, $P_i(s) \in \overline{S_i}$ for all i. Because S_i are relatively closed with respect to $\mathcal{L}_m(\mathbf{M}_i)$, i.e., $S_i = \overline{S_i} \cap \mathcal{L}_m(\mathbf{M}_i)$, we have that $P_i(s) \in S_i$, implying $s \in P_i^{-1}(S_i)$ for all i and therefore $s \in S_1 \parallel S_2$.

($\Leftarrow$) Let us now show that if (1) and (2) are valid, then the observability property at hand also holds.

Suppose observability is not valid for some S_i. Then, for some i, there are strings $s, s' \in \overline{S_i}$, with $P_{i\mathrm{s}}(s) = P_{i\mathrm{s}}(s')$, and $\sigma \in \Sigma^{\mathrm{s}}\backslash\Sigma_{i,\mathrm{c}}$ such that $s'\sigma \in \overline{S_i}$, $s\sigma \in \mathcal{L}(\mathbf{M}_i)$ and $s\sigma \notin \overline{S_i}$. But then, because $\sigma \in \Sigma^{\mathrm{s}}\backslash\Sigma_{i,\mathrm{c}}$, we have that $\sigma \in f_i(s)$, so $s\sigma \in \mathcal{L}(f_i/\mathbf{M}_i)$ since $s\sigma \in \mathcal{L}(\mathbf{M}_i)$. Moreover, because $P_{1\mathrm{s}}(S_1) = P_{2\mathrm{s}}(S_2) \Rightarrow P_{1\mathrm{s}}(\overline{S_1}) = P_{1\mathrm{s}}(\overline{S_2})$, and $s'\sigma \in \overline{S_i}$, and $\sigma \in \Sigma^{\mathrm{s}}$, there is a string $\hat{s} \in \Sigma_j^*$ such that $\hat{s}\sigma \in \overline{S_j}$ and $P_{j\mathrm{s}}(\hat{s}) = P_{i\mathrm{s}}(s')$. Thus, $\sigma \in f_j(\hat{s})$; plus, we know $\overline{S_j} \subseteq \mathcal{L}(\mathbf{M}_j)$, implying $\hat{s}\sigma \in \mathcal{L}(f_j/\mathbf{M}_j)$. Since $P_{j\mathrm{s}}(\hat{s}) = P_{i\mathrm{s}}(s') = P_{i\mathrm{s}}(s)$, we have that $P_{i\mathrm{s}}(s\sigma) = P_{j\mathrm{s}}(\hat{s}\sigma)$, so $P_{i\mathrm{s}}^{-1}(s\sigma) \cap P_{j\mathrm{s}}^{-1}(\hat{s}\sigma) \neq \emptyset$, which implies $\exists u \in \Sigma^*$ such that $P_i(u) = s\sigma$ and $P_j(u) = \hat{s}\sigma$. Hence, $u \in \mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2)$, but $u \notin \overline{S_1} \parallel \overline{S_2}$, as $P_i(u) = s\sigma \notin \overline{S_i}$. □

Corollary 1. In the context of Definition 5, the global behaviour of the plant in closed loop with the supervisors f_i is nonblocking, that is,

$$\mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2) = \overline{\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2)},$$

if and only if S_1 and S_2 are nonconflicting and S_i observable with respect to $\mathcal{L}(\mathbf{M}_i)$, $\Sigma^{\mathrm{s}}\backslash\Sigma^{\mathrm{s}}_{i,\mathrm{c}}$ and $P_{i\mathrm{s}} : \Sigma_i^* \to (\Sigma^{\mathrm{s}})^*$.

Proof. This result follows directly from the definition of conflict and from Lem. 1. □

Theorem 2. Given the premisses of Thm. 1, if S_1 and S_2 are nonconflicting we have that

$$S_1 \parallel S_2 = \sup\mathcal{C}(K_1 \parallel K_2),$$

where $\sup\mathcal{C}$ is taken with respect to $\mathcal{L}(\mathbf{M}_1 \parallel \mathbf{M}_2)$ and $\Sigma^{\ell}_{1,\mathrm{uc}} \cup \Sigma^{\ell}_{2,\mathrm{uc}}$.

Proof. Let us denote by S the composition $S_1 \parallel S_2$. The languages S_1 and S_2 are nonconflicting, so $\overline{S} = \overline{S_1} \parallel \overline{S_2}$. Moreover, each language S_i is controllable with respect to $\mathcal{L}(\mathbf{M}_i)$ and $\Sigma^{\ell}_{i,\mathrm{uc}}$, because $\mathbf{S}_i$ is the output of the CSynth function used in Negotiation. Thus, since the synchronisation in $\overline{S_1} \parallel \overline{S_2}$ only occurs over shared events, which are considered controllable, it is straightforward to show that S is controllable with respect to $\mathcal{L}(\mathbf{M}_1 \parallel \mathbf{M}_2)$ and $\Sigma^{\ell}_{1,\mathrm{uc}} \cup \Sigma^{\ell}_{2,\mathrm{uc}}$. Then, for there is no language L such that $S \subset L \subseteq S$, we have that $S = \sup\mathcal{C}(S)$. Finally, by Thm. 1, it follows that $S = \sup\mathcal{C}(K_1 \parallel K_2)$. □

Proposition 1. For trim automata $\mathbf{S}_1$ and $\mathbf{S}_2$ as in Thm. 1, if their marked languages S_1 and S_2 are unambiguous with respect to Σ^{s} and the natural projection $P_{i\mathrm{s}} : \Sigma_i^* \to (\Sigma^{\mathrm{s}})^*$, then these languages are nonconflicting.

Proof. For any language S_i, it is always true (and trivial to show) that $\overline{S_1 \parallel S_2} \subseteq \overline{S_1} \parallel \overline{S_2}$. Let us then prove the other direction, i.e., that $\overline{S_1} \parallel \overline{S_2} \subseteq \overline{S_1 \parallel S_2}$. Once more, we use the definitions of P_i and P_{s} as in the proof of Lemma 1.

Let t be any string in $\overline{S_1} \parallel \overline{S_2} = P_1^{-1}(\overline{S_1}) \cap P_2^{-1}(\overline{S_2})$. Then, for any $i \in \{1,2\}$ there exists $u_i \in \Sigma_i^*$ such that $t_i u_i \in S_i$, where $t_i = P_i(t)$. If $P_{1\mathrm{s}}(u_1) = P_{2\mathrm{s}}(u_2) = \epsilon$, take $w \in \Sigma^*$ such that $w = u_1 u_2$. Then, $P_i(tw) = t_i u_i \in S_i$, so $tw \in S_1 \parallel S_2$ and hence $t \in \overline{S_1 \parallel S_2}$.

Else, there is i such that $P_{i\mathrm{s}}(u_i) \neq \epsilon$. Now, because $P_{1\mathrm{s}}(S_1) = P_{2\mathrm{s}}(S_2)$, there exist $\hat{t}_j$ and $\hat{u}_j$ such that $P_{j\mathrm{s}}(\hat{t}_j) = P_{i\mathrm{s}}(t_i)$, $P_{j\mathrm{s}}(\hat{u}_j) = P_{i\mathrm{s}}(u_i)$, and $\hat{t}_j\hat{u}_j \in S_j$ (note that we might have $\hat{t}_j \neq t_j = P_j(t)$). Thus, in summary, we have that $t_j \in \overline{S_j}$ (from the initial assumption), $\hat{t}_j \in \overline{S_j}$, and $P_{j\mathrm{s}}(\hat{t}_j) = [P_{i\mathrm{s}}(t_i) = P_{\mathrm{s}}(t) =] P_{j\mathrm{s}}(t_j)$. Because $P_{j\mathrm{s}}(\hat{u}_j) = P_{i\mathrm{s}}(u_i) \neq \epsilon$, there exist $\alpha \in \Sigma^{\mathrm{s}}$ and $\hat{v}_j < \hat{u}_j$ such that $P_{j\mathrm{s}}(\hat{v}_j) = \epsilon$ and $\hat{v}_j\alpha \leq \hat{u}_j$. Then, $\hat{t}_j\hat{v}_j \in \overline{S_j}$, $P_{j\mathrm{s}}(\hat{t}_j\hat{v}_j) = [P_{j\mathrm{s}}(\hat{t}_j) =] P_{j\mathrm{s}}(t_j)$, and $\hat{t}_j\hat{v}_j\alpha \in \overline{S}_j$. Therefore, by condition (1) of unambiguity we have that there exists $v_j \in (\Sigma^{\ell}_j)^*$ such that $t_j v_j \alpha \in \overline{S_j}$. Note that, since $P_{j\mathrm{s}}(\hat{u}_j) \neq \epsilon$, there exist $\lambda \in \Sigma^{\mathrm{s}}$ and $v'_j \in \Sigma_j^*$ such that

$$v'_j\lambda \leq \hat{u}_j \text{ and } P_{j\mathrm{s}}(v'_j\lambda) = P_{j\mathrm{s}}(\hat{u}_j), \qquad (*)$$

i.e., there is an event λ which is the last shared event that occurs in $\hat{u}_j$. It may also be that $v'_j = \hat{v}_j$ and $\lambda = \alpha$. Nonetheless, if $P_{j\mathrm{s}}(\hat{u}_j) \neq \alpha$ — i.e., $\hat{v}_j < v'_j$ — note that we can reapply the same argument as before (where we use the unambiguity property) as many times as the length of $P_{j\mathrm{s}}(\hat{u}_j)$ minus 1 — i.e., in total, we apply condition (1) from unambiguity as many times as there are shared events in u_i (or, equivalently, in $\hat{u}_j$). This implies that there exists $\tilde{v}_j \in \Sigma_j^*$ such that $P_{j\mathrm{s}}(\tilde{v}_j) = P_{j\mathrm{s}}(v'_j)$ and $t_j\tilde{v}_j\lambda \in \overline{S_j}$. From $(*)$, we have that there exists $w' \in (\Sigma^{\ell}_j)^*$ such that $v'_j\lambda w' = \hat{u}_j$, which implies $\hat{t}_j v'_j \lambda w' = \hat{t}_j\hat{u}_j \in S_j$. But $P_{j\mathrm{s}}(\hat{t}_j v'_j\lambda w') = P_{j\mathrm{s}}(\hat{t}_j v'_j\lambda) = P_{j\mathrm{s}}(t_j\tilde{v}_j\lambda)$; thus, by condition (2) of unambiguity there exists $w \in (\Sigma^{\ell}_j)^*$ such that $t_j\tilde{v}_j\lambda w \in S_j$, implying $P_{j\mathrm{s}}(\tilde{v}_j\lambda w) = P_{j\mathrm{s}}(v'_j\lambda) = P_{j\mathrm{s}}(\hat{u}_j) = P_{i\mathrm{s}}(u_i)$, so $P_i^{-1}(u_i) \cap P_j^{-1}(\tilde{v}_j\lambda w) \neq \emptyset$ and hence there exists $u \in \Sigma^*$ such that $P_i(u) = u_i$ and $P_j(u) = \tilde{v}_j\lambda w$. Then, we have $P_i(tu) = t_i u_i \in S_i$ and $P_j(tu) = P_j(t_j\tilde{v}_j\lambda w) \in S_j$, which implies $tu \in S_1 \parallel S_2$ and so $t \in \overline{S_1 \parallel S_2}$. □

Remark 4. Let $\tilde{\mathbf{S}}_i$ be an automaton over Σ_i such that $\tilde{S}_i \subseteq K_i$, $\mathcal{L}(\tilde{\mathbf{S}}_i) \subseteq \mathcal{L}(\mathbf{K}_i)$ and such that the partition of Σ_i with respect to controllability and shared/local events is the same as we defined in Sec. 3.1. Then $\sup\mathcal{C}(\tilde{S}_1 \parallel \tilde{S}_2) = \sup\mathcal{C}(\hat{S}_1 \parallel \hat{S}_2)$, where $(\hat{\mathbf{S}}_1, \hat{\mathbf{S}}_2) = \textsc{Negotiation}(\tilde{\mathbf{S}}_1, \tilde{\mathbf{S}}_2)$ and $\sup\mathcal{C}$ is taken with respect to $\mathcal{L}(\mathbf{M}_1 \parallel \mathbf{M}_2)$ and $\Sigma^{\ell}_{1,\mathrm{uc}} \cup \Sigma^{\ell}_{2,\mathrm{uc}}$.

Proof. Just note that, for this scenario, the same proof as the one used for Thm. 1 holds. □

Theorem 3. Consider a plant composed of two subsystems $\mathbf{M}_i$ with plantified specifications $\mathbf{K}_i$, as in Sec. 3.1. Let $(\mathbf{S}_1, \mathbf{S}_2) = \mathrm{CBSS}(\mathbf{K}_1, \mathbf{K}_2)$. Further, let f_i be the local supervisors defined from $\mathbf{S}_i$ via Def. 5. Then, it holds that

(1) $\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2) \in \mathcal{C}(K_1 \parallel K_2)$, and
(2) $\mathcal{L}(f_1/\mathbf{M}_1) \parallel \mathcal{L}(f_2/\mathbf{M}_2) = \overline{\mathcal{L}_m(f_1/\mathbf{M}_1) \parallel \mathcal{L}_m(f_2/\mathbf{M}_2)}$.

Proof. First, note the following remarks:

- ▷ The output of EnforceUnambiguity in general may not be a trim automaton, for the composition of $\tilde{\mathbf{\Lambda}}$ with $\mathbf{O}_{\mathbf{\Lambda}}$, which is trimmed, only removes markings and transitions by shared events from the first automaton. Nevertheless, the fixed point of CBSS is also a fixed point for (the last call of) Negotiation. Thus, each automaton $\mathbf{S}_i$ is trim and thus nonblocking; moreover, it is controllable with respect to $\mathcal{L}(\mathbf{M}_i)$ and $\Sigma^{\ell}_{i,\mathrm{uc}}$, while the supervisor f_i from Def. 5 cannot control events in $\Sigma^{\ell}_{i,\mathrm{uc}} \dot{\cup} (\Sigma^{\mathrm{s}}\backslash\Sigma^{\mathrm{s}}_{i,\mathrm{c}})$. Therefore, the same proof as the one used for Lem. 1 also holds if, instead of taking languages S_i such that $(\mathbf{S}_1, \mathbf{S}_2) = \textsc{Negotiation}(\mathbf{K}_1, \mathbf{K}_2)$, we take S_i such that $(\mathbf{S}_1, \mathbf{S}_2) = \mathrm{CBSS}(\mathbf{K}_1, \mathbf{K}_2)$. Hence, Lem. 1 and Cor. 1 are also valid for such languages.
- ▷ From the definitions of observability, unambiguity and relative unambiguity we have that, given a language $L_i \subseteq K_i$, if another language $L \subseteq L_i$ is relatively unambiguous with respect to L_i, alphabets Σ^{s} and $\Sigma^{\mathrm{s}}_{i,\mathrm{uc}} = \Sigma^{\mathrm{s}}\backslash\Sigma^{\mathrm{s}}_{i,\mathrm{c}}$, $\mathcal{L}(\mathbf{M}_i)$ and the projection $P_{i\mathrm{s}} : \Sigma_i^* \to (\Sigma^{\mathrm{s}})^*$, then in particular L is: (i) observable with respect to $\mathcal{L}(\mathbf{M}_i)$, $\Sigma^{\mathrm{s}}_{i,\mathrm{uc}}$ and $P_{i\mathrm{s}}$; and (ii) unambiguous with respect to Σ^{s} and $P_{i\mathrm{s}}$.

- ▷ As the fixed point for CBSS is also a fixed point for (the last call of) EnforceUnambiguity, and this procedure enforces relative unambiguity, as explained in Sec. 5.3, then we have that each language S_i is observable with respect to $\mathcal{L}(\mathbf{M}_i)$, $\Sigma^{\mathrm{s}}_{i,\mathrm{uc}}$ and $P_{i\mathrm{s}}$, and unambiguous with respect to Σ^{s} and $P_{i\mathrm{s}}$. Furthermore, being unambiguous, by Prop. 1 we also have that S_1 and S_2 are nonconflicting.
- ▷ Each call of the EnforceUnambiguity procedure in the CBSS algorithm only removes states, transitions and marking from the automaton given as input, so if for this input we have that its marked language is contained or equal to K_i, then the same is also true for the output. This fact together with Remark 1 implies that $S_i \subseteq K_i$ and, therefore, that $S = S_1 \parallel S_2 \subseteq K_1 \parallel K_2 = K$. Because S_1 and S_2 are nonconflicting, we can use the same arguments as in the proof of Thm. 2 to show that S is controllable with respect to $\mathcal{L}(\mathbf{M}_1 \parallel \mathbf{M}_2)$ and $\Sigma^{\ell}_{1,\mathrm{uc}} \cup \Sigma^{\ell}_{2,\mathrm{uc}}$, i.e., $S \in \mathcal{C}(K)$.

Now, because $S \in \mathcal{C}(K)$ and because each language S_i is observable as stated above, we can apply Lem. 1 to prove condition (1) from Thm. 3. Finally, given that S_i are nonconflicting and observable, as explained above, from Cor. 1 we prove condition (2). □