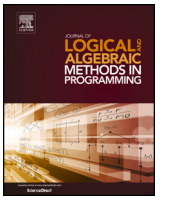
J. Log. Algebraic Methods Program. 141 (2024) 100998

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



Advancing orchestration synthesis for contract automata

Davide Basile *, Maurice H. ter Beek

Formal Methods and Tools lab, CNR–ISTI, Via G. Moruzzi 1, Pisa, 56124, Italy

A B S T R A C T

Contract automata allow to formally define the behaviour of service contracts in terms of service offers and requests, some of which are moreover optional and some of which are necessary. A composition of contracts is said to be in agreement if all service requests are matched by corresponding offers. Whenever a composition of contracts is not in agreement, it can be refined to reach an agreement using the orchestration synthesis algorithm. This algorithm is a variant of the synthesis algorithm used in supervisory control theory and it is based on the fact that optional transitions are controllable, whereas necessary transitions are at most semi-controllable and cannot always be controlled. In this paper, we present advancements of the orchestration synthesis for contract automata. Notably, we identify the existing limits of the orchestration synthesis and propose a novel orchestration synthesis along with additional constructs to enhance the expressiveness and scalability of contract automata. The proposed advancements have been implemented and experimented on two case studies, one of which originates from the railway domain and the other is a card game.

1. Introduction

Orchestrations of services describe how control and data exchanges are coordinated in distributed service-based applications and systems. Their principled design is identified in the Service Computing Manifesto [1] in 2017 as one of the primary research challenges for the next 10 years, by pointing out that "Service systems have so far been built without an adequate rigorous foundation that would enable reasoning about them" and, moreover, that "The design of service systems should build upon a formal model of services".

The problem of synthesising well-behaving orchestrations of services can be viewed as a specific instance of the more general problem of synthesising strategies in games [2,3]. This can be solved using refined algorithms from supervisory control for discrete event systems [4,5], which have well-established relationships with reactive systems synthesis [6], parity games [7], automated behaviour composition [8] and automated planning [9].

Contract automata are a specific type of finite state automata, used to formally define the behaviour of service contracts by expressing contracts in terms of offers and requests [10]. When multiple contracts are composed, they are said to be in agreement if all service requests from one contract are matched by another contract's corresponding offers. A composition of contracts that is not in agreement can automatically be refined to reach an agreement by means of the orchestration synthesis algorithm, which is a variant of the synthesis algorithm used in supervisory control theory. This orchestration synthesis algorithm for contract automata is defined in [11,2].

The classical algorithm for synthesising a most permissive controller distinguishes transitions whose controllability is invariant [4,5] (i.e., a transition is either controllable or uncontrollable, where controllable transitions can be blocked by the controller). In service contracts, instead, the controllability of certain transitions (called semi-controllable) may vary depending on specific conditions on the orchestration of contracts [2]. The contract automata library `CATLib` [12] implements contract automata and their operations (e.g., composition and synthesis).

* Corresponding author.
E-mail address: davide.basile@isti.cnr.it (D. Basile).

https://doi.org/10.1016/j.jlamp.2024.100998
Received 2 January 2024; Received in revised form 10 June 2024; Accepted 11 June 2024

Available online 15 June 2024

This paper is an extension of [13], where several research challenges for contract automata are presented. We start by proposing an intuitive interpretation of semi-controllable transitions, and show how the current notion of orchestration synthesis (which we term *conditional* orchestration) is too abstract to capture this intended interpretation. Subsequently, we refine the conditional orchestration by strengthening the conditions of semi-controllability. We provide several examples to illustrate the differences between the refined definition and the original definition. The various notions of semi-controllability lead to different sets of conditional orchestrations, which we present in Fig. 5 together with an example for each level of the orchestration hierarchy depicted. We present a number of research questions for advancing the orchestration synthesis of contract automata, originally proposed in [13], and address them. Specifically, [13] is extended as follows.

- We formalise a new orchestration synthesis, called *splitting orchestration*. In a splitting orchestration, semi-controllable transitions are no longer controllable or uncontrollable depending on given conditions, but are rather split into pairs of concatenated uncontrollable and controllable transitions. We show how the splitting orchestration precisely characterises the intended intuition of semi-controllable transitions, which can only be approximated using conditional orchestrations.
- We improve the expressiveness of contract automata by introducing new constructs, with the goal of improving scalability and reducing the generated state space of a composition.
- The new notions of orchestration, together with the newly presented scalable techniques, are implemented in `CATLib` [14].
- We perform a set of experiments to compare the various orchestrations, based on two case studies, of which one is taken from the railway domain [15] and the other is a card game. The conditional orchestrations are showed to be practical heuristics approximating the splitting orchestration.

Related work At the ICE 2022 workshop, the compositionality of communicating finite state machines (CFSM) with asynchronous semantics was discussed in [16]. Also contract automata are composable, enabling the modelling of systems of systems. Moreover, under certain specific conditions that were presented at the 2014 edition of ICE [17,18], an orchestration of contract automata can be translated into a choreography of synchronous or asynchronous CFSM. The relation between multiparty session types and CFSM is discussed in [19]. Therefore, contract automata can be related to multiparty session types by exploiting their common relation with CFSM [17–19].

The contract automata approach is closer to [20], where behavioural types are expressed as finite state automata of `Mungo`, called typestates [21]. Similarly to `CARE`, the runtime environment for contract automata [22,23], in `Mungo` finite state automata are used as behaviour assigned to Java classes (one automaton per class), with transition labels corresponding to methods of the classes. A tool to translate typestates into automata was presented at ICE 2020 [24]. `CATApp` is a graphical front-end tool for designing contract automata [25]. A tool similar to `Mungo` is `JaTyC` (Java Typestate Checker) [26].

The refined definition of semi-controllability we present in this paper closely aligns with the notion of weak receptiveness in team automata [27,28]. However, the research questions addressed in this paper are primarily related to the problem of synthesising an orchestration of services and as such are not directly relevant to team automata (cf. [29] for a brief comparison of contract automata with team automata).

Differently from the semi-controllability for orchestrations, a distinct notion of semi-controllability has been studied in [2,30] for choreographies of services. Finally, while a runtime environment for the orchestration of services has been proposed in [22], this has yet to be realised for the case of choreographies, which could result in improvements in the notion of semi-controllability for choreographies.

Outline We start by providing some background on contract automata, orchestration and most permissive controller synthesis in Section 2. We discuss current limits of the orchestration synthesis of contract automata and the intended intuition of semi-controllability in Section 3. In Section 4, we present several research questions for orchestration synthesis of contract automata. In Section 5, we address all the proposed research questions. We conclude in Section 6. Appendix A contains the proofs of two results from Section 5.

2. Background

We begin by formally introducing contract automata and their synthesis operation. Contract automata are a type of finite state automata that use a partitioned alphabet of actions. A Contract Automaton (CA) can model either a single service or a composition of multiple services that perform actions. The number of services in a CA is called its rank. If the rank of a CA is 1, then the contract is referred to as a principal (i.e., a single service).

The labels of a CA are vectors of atomic elements called actions. Actions are categorised as either requests (prefixed by?), offers (prefixed by !) or idle actions (represented by a distinguished symbol $-$). Requests and offers belong to the sets R and O, respectively, and they are pairwise disjoint. The states of a CA are vectors of atomic elements called basic states. Labels are restricted to requests, offers or matches. In a request (resp. offer) label there is a single request (resp. offer) action and all other actions are idle. In a match label there is a single pair of request and offer actions that match, and all other actions are idle. The length of the vectors of states and labels is equal to the rank of the CA. For example, the label $[!a, ?a, -, -]$ is a match where the request action $?a$ is matched by the offer action $!a$, and all other actions are idle. Note the difference between a request label (e.g., $[?a, -]$) and a request action (e.g., $?a$). A transition may also be called a request, offer or match according to its label.

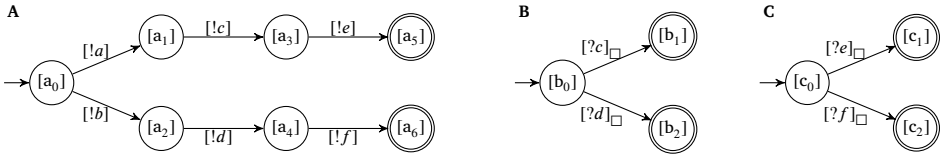


Fig. 1. Contracts of **A**lice, **B**ob and **C**arl.

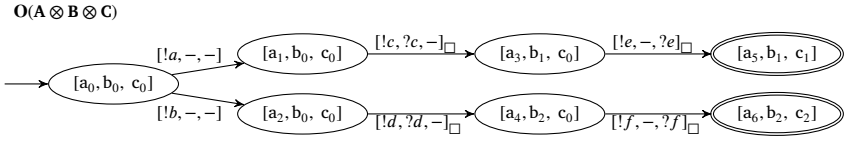


Fig. 2. Orchestration **O**(**A** ⊗ **B** ⊗ **C**) of **A**lice ⊗ **B**ob ⊗ **C**arl.

Example 1. Fig. 1 depicts three principal contracts, whilst Fig. 2 depicts a contract of rank 3. The transition $([a_0, b_0, c_0], [!a, -, -], [a_1, b_0, c_0])$ of the contract in Fig. 2 has the offer label $[!a, -, -]$, where the first service performs the offer $!a$ and the second and third services are idle.

The goal of each service is to reach an accepting (*final*) state such that all its request (and possibly offer) actions are matched. Transitions are equipped with *modalities*, i.e., *necessary* ($\Box$) and *optional* ($\bigcirc$) transitions, respectively.[1] Optional transitions are controllable, whereas necessary transitions can be uncontrollable (called *urgent* necessary transitions) or semi-controllable (called *lazy* necessary transitions). Controllable transitions can be blocked by the controller, whereas uncontrollable transitions cannot be blocked. Semi-controllable transitions are transitions that are either controllable or uncontrollable depending on the given conditions. Intuitively, an urgent transition cannot be delayed, whereas a lazy one can. In other words, the scheduling of necessary lazy requests is controlled by the orchestrator, whilst this is not the case for urgent necessary requests, the scheduling of which is uncontrollable.

The resulting formalism is called *Modal Service Contract Automata* (MSCA). In the following definition, given a vector $\vec{a}$, its ith element is denoted by $\vec{a}_{(i)}$.

Definition 1 *(MSCA).* Given a finite set of states $Q = \{q_1, q_2, \dots\}$, an MSCA $\mathcal{A}$ of rank n is a tuple $\langle Q, \vec{q}_0, A^r, A^o, T^{\Box_u}, T^{\Box_l}, T^{\bigcirc}, F\rangle$, with set of states $Q = Q_1 \times \dots \times Q_n \subseteq Q^n$, initial state $\vec{q}_0 \in Q$, set of requests $A^r \subseteq \mathsf{R}$, set of offers $A^o \subseteq \mathsf{O}$, set of final states $F \subseteq Q$, set of *optional* transitions $T^{\bigcirc}$, set of *urgent* necessary transitions $T^{\Box_u}$, set of *lazy* necessary transitions $T^{\Box_l}$. The sets $T^{\bigcirc}$, $T^{\Box_u}$ and $T^{\Box_l}$ are pairwise disjoint, $T^{\Box} = T^{\Box_u} \cup T^{\Box_l}$ is the set of *necessary* transitions, and $T = (T^{\Box_u} \cup T^{\Box_l} \cup T^{\bigcirc}) \subseteq Q \times A \times Q$ is the set of transitions where $A \subseteq (A^r \cup A^o \cup \{-\})^n$. Furthermore, given $t = (\vec{q}, \vec{a}, \vec{q}') \in T$: i) $\vec{a}$ is either a request, an offer or a match; ii) if $\vec{a}$ is an offer, then $t \in T^{\bigcirc}$; and iii) $\forall i \in 1 \dots n,\ \vec{a}_{(i)} = -$ implies $\vec{q}_{(i)} = \vec{q}'_{(i)}$.

Composition of services is rendered through the composition of their MSCA models by means of the *composition operator* $\otimes$ [11, Definition 5][10, Definition 2.5], which is a variant of the synchronous product. This operator basically interleaves or matches the transitions of the component MSCA, but, whenever two component MSCA are enabled to execute their respective request/offer action, then the match is forced to happen. Moreover, a match involving a necessary transition of an operand is itself necessary. The rank of the composed MSCA is the sum of the ranks of its operands. The vectors of states and actions of the composed MSCA are built from the vectors of states and actions of the component MSCA, respectively. In this paper, we will mainly consider principal contracts and compositions of principals, which will be automatically refined into orchestrations (as shown in Fig. 4). However, it is important to note that contracts can be created by composing contracts with a rank of one or higher. Moreover, for brevity, we will often speak of contract automata (CA) rather than MSCA.

Example 2. Fig. 2 depicts a portion of the composition of the three principal contracts from Fig. 1.

In a composition of MSCA, typically various properties are analysed. We are especially interested in *agreement*. The property of agreement requires to match all request actions of all services, whereas offer actions can remain unmatched. In other words, labels of CA satisfying agreement are either match labels or offer labels. We will also use a stricter property known as *strong agreement*. Match transitions satisfy strong agreement, whereas offer and request transitions do not.

CA support the synthesis of the most permissive controller (mpc) known from the theory of supervisory control of discrete event systems [4,33], where a finite state automaton model of a *supervisory controller* is synthesised from given (component) finite state automata that are composed. The synthesised automaton, if successfully generated (i.e., non-empty), is such that it is *non-blocking*, *controllable* and *maximally permissive*.

[1] Originally, in [11], the optional modality was called permitted and denoted with $\Diamond$. Since in contract automata the two modalities are a partition, the terminology has been updated to avoid confusion with modal transition systems [31,32], where $\Box \subseteq \Diamond$.

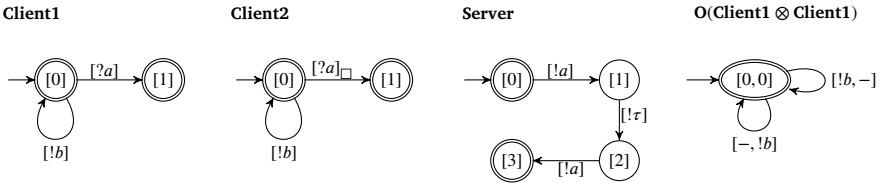


Fig. 3. Contracts of **Client1**, **Client2** and **Server**, and orchestration **O**(**Client1** ⊗ **Client1**).

An automaton is said to be *non-blocking* if, from each state, at least one of the *final states* (distinguished stable states that represent completed 'tasks' [4]) can be reached without passing through so-called *forbidden states*, meaning that there is always a possibility to return to an accepted stable state (e.g., a final state). The synthesised automaton is said to be *controllable* when only controllable transitions are disabled. Indeed, the supervisory controller is not permitted to directly block uncontrollable transitions from occurring; the controller is only allowed to disable them by preventing controllable actions from occurring. Finally, the fact that the resulting supervisory controller is said to be *maximally permissive* (or least restrictive) means that as much behaviour of the uncontrolled system as possible is present in the controlled system without violating neither the requirements, nor controllability nor the non-blocking condition.

Example 3. Consider the automaton depicted in Fig. 2, which depicts a portion of the composition of the principal contracts shown in Fig. 1. Assume that all necessary transitions are urgent (i.e., uncontrollable), while optional transitions are, as usual, controllable. The automaton is non-blocking, as it is possible to reach a final state from each state. However, the automaton is not controllable. Indeed, from the initial state, the transitions of **B** and **C** are blocked, despite being uncontrollable. In this case, the mpc of $\mathbf{A} \otimes \mathbf{B} \otimes \mathbf{C}$ is the empty automaton.

Orchestration Synthesis As stated previously, optional transitions are controllable, whereas necessary transitions can be either uncontrollable (called *urgent*) or semi-controllable (called *lazy*). In the mpc synthesis (implemented in `CATLib` [2,12]), all necessary transitions are *urgent*, i.e., they are always uncontrollable. This stems from the fact that traditionally uncontrollable transitions relate to an unpredictable environment.

When synthesising an orchestration of services, all necessary transitions are instead *lazy*, i.e., they are *semi-controllable* [11,2]. A semi-controllable transition t is a transition that is either uncontrollable or controllable according to given conditions. In [2], different conditions are provided according to whether the synthesis of an orchestration or a choreography is computed. In this paper, we do not consider choreographies. Below, we denote with $\mathit{Dangling}(\mathcal{A})$ the set of states that are not reachable from the initial state or cannot reach any final state. Furthermore, given two MSCA $\mathcal{A}$ and $\mathcal{A}'$, we say that $\mathcal{A}'$ is a *sub-automaton* of $\mathcal{A}$, denoted by $\mathcal{A}' \subseteq \mathcal{A}$, whenever the components of $\mathcal{A}'$ are included in the corresponding ones of $\mathcal{A}$.

In Definition 2, the automaton $\mathcal{A}'$ represents an intermediate refinement of $\mathcal{A}$ (the starting composition) which occurs during an iteration of the synthesis process. Intuitively, the semi-controllable transition t of $\mathcal{A}$ is controllable in $\mathcal{A}'$ because there is another transition t' in $\mathcal{A}'$ matching the same request from the same service in the same local state, and the source and target states of t' are not dangling. Otherwise, if there is no such transition t' in $\mathcal{A}'$, then t is uncontrollable.

In other words, the controllability of t in $\mathcal{A}'$ relies on the presence of a corresponding transition t' within $\mathcal{A}'$ itself. If such a matching transition t' does not exist in $\mathcal{A}'$, then t is deemed uncontrollable.

Definition 2 *(Controllability).* Let $\mathcal{A}$ be an MSCA and let $t = (\vec{q}_1, \vec{a}_1, \vec{q}_1{}') \in T_{\mathcal{A}}$. Then:

- if $t \in T_{\mathcal{A}}^{\bigcirc}$, then t is *controllable* (in $\mathcal{A}$);
- if $t \in T_{\mathcal{A}}^{\Box_u}$, then t is *uncontrollable* (in $\mathcal{A}$);
- if $t \in T_{\mathcal{A}}^{\Box_l}$, then t is *semi-controllable* (in $\mathcal{A}$).

Moreover, given $\mathcal{A}' \subseteq \mathcal{A}$, if t is semi-controllable and $\exists\, t' = (\vec{q}_2, \vec{a}_2, \vec{q}_2{}') \in T_{\mathcal{A}'}^{\Box}$ in $\mathcal{A}'$ such that $\vec{a}_2$ is a match, $\vec{q}_2, \vec{q}_2{}' \notin \mathit{Dangling}(\mathcal{A}')$, $\vec{q}_{1(i)} = \vec{q}_{2(i)}$, and $\vec{a}_{1(i)} = \vec{a}_{2(i)} = ?a$ for some $i \in 0 \dots \mathit{rank}(\mathcal{A})$, then t is *controllable* in $\mathcal{A}'$ (via t'). Otherwise, t is *uncontrollable* in $\mathcal{A}'$.

We use the standard interpretation of optional/controllable and urgent/ uncontrollable transitions [4,33]. In the upcoming sections, we will discuss different interpretations of the concept of semi-controllability. We remark that the orchestration synthesis defined below does not support urgent transitions. The orchestration synthesis, defined in Definition 3, involves an iterative refinement of the initial automaton $\mathcal{A}$ (i.e., the composition of contracts). In each iteration, transitions are selectively pruned, and a set R of forbidden states is updated accordingly. A transition t is pruned under one of two conditions: if it is a request (thus violating the agreement property enforced by the orchestration), or if the target state of t belongs to the set R computed up to that point. During the first iteration, all request transitions, including both lazy and optional ones, are pruned.

Note that in Definition 2, it is not required for t and t' to be distinct. This implies that during the synthesis process, a semi-controllable match transition t can switch from being controllable to uncontrollable only after it has been pruned in a previous

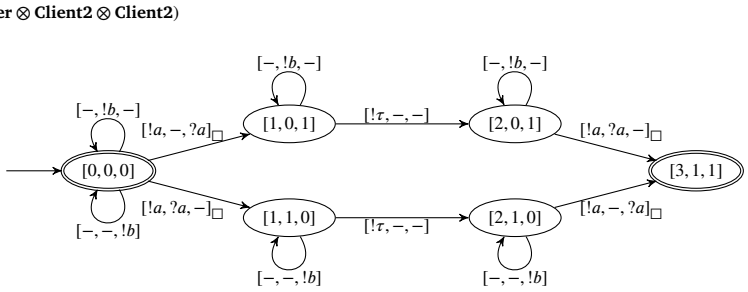


Fig. 4. Orchestration **O(Server ⊗ Client2 ⊗ Client2)**.

iteration. To clarify further, a semi-controllable match transition t can switch its controllability status from controllable to uncontrollable only when t is absent in the sub-automaton $\mathcal{A}'$ during the current iteration. If t is present in $\mathcal{A}'$ (i.e., it has not been pruned thus far), then, according to Definition 2, t is considered semi-controllable and controllable within $\mathcal{A}'$ via t itself. It is important to note that these considerations are applicable only if t is a match. Additionally, it is never the case that a semi-controllable transition t switches from uncontrollable to controllable since transitions are only removed during the synthesis process and are never added back. Example 5 presented below will illustrate the synthesis of orchestration with semi-controllable transitions.

The set R of forbidden states is updated at each iteration by adding source states of uncontrollable transitions and dangling states of the refined automaton in the current iteration. Specifically, when the synthesis process eliminates all transitions t' that satisfy the conditions for rendering the semi-controllable transition t controllable via t', then t becomes uncontrollable within the sub-automaton in the current iteration. It is worth noting that even if t was previously pruned in an earlier iteration, its source state $\vec{q}_1$ might still be reachable in the sub-automaton of the current iteration. Consequently, $\vec{q}_1$ is added to the set R. In the subsequent iteration, all transitions with target state $\vec{q}_1$ will be pruned. This pruning of transitions whose target is $\vec{q}_1$ can potentially render another previously pruned semi-controllable transition as uncontrollable, thereby adding its source state to the updated set R. This refinement process continues until no further transitions are pruned, and no additional states are added to R. The resulting refined automaton obtained at the end of the synthesis process represents the orchestration automaton.

In this paper, we will present different orchestration synthesis algorithms. We term the orchestration synthesis introduced in [2] as *conditional* orchestration synthesis (we refer to it simply as orchestration synthesis when it is clear from the context). This reflects the fact that a condition determines whether a semi-controllable transition is controllable or uncontrollable. The conditional orchestration synthesis enforcing agreement is defined below.

Definition 3 *(Conditional orchestration synthesis).* Let $\mathcal{A}$ be an MSCA and let $\mathcal{K}_0 = \mathcal{A}$ and $R_0 = \mathit{Dangling}(\mathcal{K}_0)$. We let the *orchestration synthesis function* $f_o : \mathit{MSCA} \times 2^Q \to \mathit{MSCA} \times 2^Q$ be defined as follows:

$$
\begin{aligned}
&f_o(\mathcal{K}_{i-1}, R_{i-1}) = (\mathcal{K}_i, R_i), \text{ with}\\
&\quad T_{\mathcal{K}_i} \;=\; T_{\mathcal{K}_{i-1}} \setminus \{\, (\vec{q} \to \vec{q}\,') = t \in T_{\mathcal{K}_{i-1}} \mid (\vec{q}\,' \in R_{i-1} \vee t \textit{ is a request}) \,\}\\
&\quad R_i \;=\; R_{i-1} \cup \{\, \vec{q} \mid (\vec{q} \to) \in T_{\mathcal{A}}^{\square_l} \textit{ is uncontrollable in } \mathcal{K}_i \,\} \cup \mathit{Dangling}(\mathcal{K}_i)
\end{aligned}
$$

The orchestration automaton is obtained from the fixpoint of the function f_o. In the rest of the paper, if not stated otherwise, all necessary transitions in the examples are lazy (cf. Definition 1); for brevity and less cluttering in the figures, we denote them by $\square$ rather than $\square_l$.

Example 4. We provide an illustrative example to underline the differences between optional transitions and necessary transitions. Fig. 3 shows two client contracts and a server contract. When all actions of the client contract are optional (**Client1**), there exists an orchestration of the composition of two **Client1** contracts, also depicted in Fig. 3 (**O(Client1 ⊗ Client1)**). Indeed the (transition labelled with the) request $?a$ is optional and can be removed to obtain the orchestration. If, instead, the request $?a$ were necessary (**Client2**), then there would be no orchestration for the composition of two **Client2** contracts, because the necessary request is never matched by a corresponding offer.

Mpc synthesis The orchestration synthesis is a variant of the mpc synthesis. While in the orchestration synthesis all necessary transitions are lazy (i.e., semi-controllable), in the mpc synthesis all necessary transitions are urgent (i.e., uncontrollable). In the mpc synthesis, at each step i, the algorithm prunes from $\mathcal{K}_{i-1}$ in a backwards fashion transitions with a target state in R_{i-1} or a forbidden source state. Forbidden states are, for example, source states of urgent transitions violating agreement. The set R_i is obtained by adding to R_{i-1} dangling states in $\mathcal{K}_i$ and source states of urgent transitions of $\mathcal{A}$ with target state in R_{i-1}. The mpc synthesis is formalised next.

Definition 4 *(Mpc synthesis, adapted from [4]).* Let $\mathcal{A}$ be an MSCA, and let $\mathcal{K}_0 = \mathcal{A}$ and $R_0 = \mathit{Dangling}(\mathcal{K}_0)$. We let the *synthesis function* $f : \mathit{MSCA} \times 2^Q \to \mathit{MSCA} \times 2^Q$ be defined as follows:

$$
\begin{aligned}
f(\mathcal{K}_{i-1}, R_{i-1}) &= (\mathcal{K}_i, R_i), \text{ with} \\
T_{\mathcal{K}_i} &= T_{\mathcal{K}_{i-1}} \setminus \{ (\vec{q} \to \vec{q}\,') \in T_{\mathcal{K}_{i-1}} \mid \vec{q}\,' \in R_{i-1} \vee \vec{q} \text{ is forbidden} \} \\
R_i &= R_{i-1} \cup \{ \vec{q} \mid (\vec{q} \to \vec{q}\,') \in T_{\mathcal{A}}^{\square}, \ \vec{q}\,' \in R_{i-1} \} \cup \mathit{Dangling}(\mathcal{K}_i)
\end{aligned}
$$

The mpc of $\mathcal{A}$, denoted by $\mathcal{K}_{\mathcal{A}}$, is obtained from the fixpoint $(\mathcal{K}_s, R_s)$ of the function f. We have that the mpc is empty, if the initial state of $\mathcal{A}$ is in R_s; otherwise, the mpc is obtained from $\mathcal{K}_s$ by removing the states in R_s.

Example 5. We continue Example 4 to illustrate the distinction between urgent and lazy necessary transitions. We consider also the **Server** contract depicted in Fig. 3. If we were to employ the traditional mpc synthesis, the clients' necessary requests ($?a$) would be treated as urgent. In such a scenario, the orchestration of the composition between two clients and the server (generated using the mpc synthesis algorithm) would be empty, indicating that no feasible orchestration exists. This is because in the composition **Server** $\otimes$ **Client2** $\otimes$ **Client2**, from the initial state, the transitions $[0,0,0] \xrightarrow{[!a,?a,-]_{\square_u}} [1,1,0]$ and $[0,0,0] \xrightarrow{[!a,-,?a]_{\square_u}} [1,0,1]$ are both possible (recall that in the composition, if two services are ready to match their corresponding offer and request, then only the match transition will be present, while the interleaving of the two request and offer transitions is not). However, in both target states of the two transitions, the server is not ready to match the other urgent request. Therefore, the transitions $[1,1,0] \xrightarrow{[-,-,?a]_{\square_u}} [1,1,1]$ and $[1,0,1] \xrightarrow{[-,?a,-]_{\square_u}} [1,1,1]$ are also present in the composition. These are transitions violating the agreement property because they are labelled by requests. Moreover, since both transitions are uncontrollable, the states $[1,1,0]$ and $[1,0,1]$ are forbidden. Both forbidden states are reachable from the initial state through uncontrollable transitions, causing the initial state $[0,0,0]$ to become forbidden, and the overall orchestration to be empty.

However, if the clients' necessary requests ($?a$) are instead considered lazy (i.e., **Client2**), then an orchestration of the composition between the server and the two clients can be achieved (computed using the orchestration synthesis). This orchestration is depicted in Fig. 4 (**O**(**Server** $\otimes$ **Client2** $\otimes$ **Client2**)). In this case, the clients take turns fulfilling their lazy necessary requests. This alternating behaviour is not possible when the necessary requests are urgent.

The orchestration in Fig. 4 is obtained after three iterations of the algorithm specified in Definition 3. Initially, $\mathcal{K}_0 = \mathcal{A} =$ **Server** $\otimes$ **Client2** $\otimes$ **Client2** and $R_0 = \mathit{Dangling}(\mathcal{A}) = \varnothing$.

With respect to the orchestration in Fig. 4, the automaton $\mathcal{A}$ contains four additional transitions that are $t_1 = [1,0,1] \xrightarrow{[-,?a,-]_{\square}} [1,1,1]$, $t_2 = [1,1,0] \xrightarrow{[-,-,?a]_{\square}} [1,1,1]$, $t_3 = [1,1,1] \xrightarrow{[!\tau,-,-]} [2,1,1]$ and $t_4 = [2,1,1] \xrightarrow{[!a,-,-]} [3,1,1]$. In the first iteration, t_1 and t_2 are removed from $\mathcal{K}_1$ because they are request transitions. We have $T_{\mathcal{K}_1} = T_{\mathcal{K}_0} \setminus \{t_1, t_2\}$. Since there are no forbidden states, these are the only two transitions that are removed during the first iteration.

Concerning the set of forbidden states R_1, we have that $t_1 \in T_{\mathcal{A}}^{\square_l}$ is controllable in $\mathcal{K}_1$ via transition $[0,0,0] \xrightarrow{[a!,a?,-]_{\square}} [1,1,0]$. Similarly, $t_2 \in T_{\mathcal{A}}^{\square_l}$ is controllable in $\mathcal{K}_1$ via $[0,0,0] \xrightarrow{[a!,-,a?]_{\square}} [1,0,1]$. Hence, the source states of t_1 and t_2 will not be added to R_1. Concerning the set $\mathit{Dangling}(\mathcal{K}_1)$, state $[1,1,1]$ was the target of only t_1 and t_2. Moreover, state $[2,1,1]$ was the target of only t_3. Therefore, states $[1,1,1]$ and $[2,1,1]$ are unreachable in $\mathcal{K}_1$. We have that $R_1 = \mathit{Dangling}(\mathcal{K}_1) = \{[1,1,1],[2,1,1]\}$. In the subsequent iteration $i = 2$, since transition t_3 has a target state in R_1, we have $T_{\mathcal{K}_2} = T_{\mathcal{K}_1} \setminus \{t_3\}$, whereas $R_2 = R_1$.

Finally, we reach the fixpoint at iteration $i = 3$, where $T_{\mathcal{K}_3} = T_{\mathcal{K}_2}$ and $R_3 = R_2$. The finalising operations for obtaining the orchestration **O** in Fig. 4 from the fixpoint $\mathcal{K}_3$ consist of removing the states in R_3, i.e., $Q_{\mathbf{O}} = Q_{\mathcal{K}_3} \setminus R_3$, and removing the remaining unreachable transitions in $\mathcal{K}_3$. In this case, transition $t_4 \in T_{\mathcal{K}_3}$ is removed from the orchestration, i.e., $T_{\mathbf{O}} = T_{\mathcal{K}_3} \setminus \{t_4\}$.

Note that, concerning urgent requests, if the $!tau$ transition were dropped from the server (i.e., the server performs two offers in sequence) then in the composition there would be no transition labelled with either $[-,-,?a]_{\square_u}$ or $[-,?a,-]_{\square_u}$, and an orchestration would exist.

In the subsequent section, we will discuss additional details and interpretations regarding the semi-controllable transitions of contract automata.

3. Refined semi-controllability

We start by introducing a refined notion of semi-controllability to be used in the orchestration synthesis, formalised below. After that, we discuss how this refined notion may assist to discard some counterintuitive orchestrations.

Definition 5 *(Refined semi-controllability).* Let $\mathcal{A}$ be an MSCA and let $t = (\vec{q}_t, \vec{a}_t, \vec{q}\,'_t) \in T_{\mathcal{A}}^{\square_l}$. Moreover, given $\mathcal{A}' \subseteq \mathcal{A}$, if $\exists\, t' = (\vec{q}_{t'}, \vec{a}_{t'}, \vec{q}\,'_{t'}) \in T_{\mathcal{A}'}^{\square_l}$ in $\mathcal{A}'$ such that the following hold:

1. $\vec{a}_{t'}$ is a match, $\vec{q}_{t'}, \vec{q}\,'_{t'} \notin \mathit{Dangling}(\mathcal{A}')$, $\vec{q}_{t(j)} = \vec{q}_{t'(j)}$, $\vec{a}_{t(j)} = \vec{a}_{t'(j)} = ?a$, for some $j \in 0 \dots \mathit{rank}(\mathcal{A})$; and
2. there exists a sequence of transitions $t_0, \dots, t_n$ of $\mathcal{A}'$ such that $\forall i \in 0 \dots n$, $t_i = (\vec{q}_i, \vec{a}_i, \vec{q}\,'_i)$ and the following hold:
   - $\vec{q}_0 = \vec{q}_t$;
   - $t_n = t'$;
   - $\vec{q}_i, \vec{q}\,'_i \notin \mathit{Dangling}(\mathcal{A}')$; and
   - if $i < n$, then $\vec{a}_{i(j)} = -$ and $\vec{q}\,'_i = \vec{q}_{i+1}$;

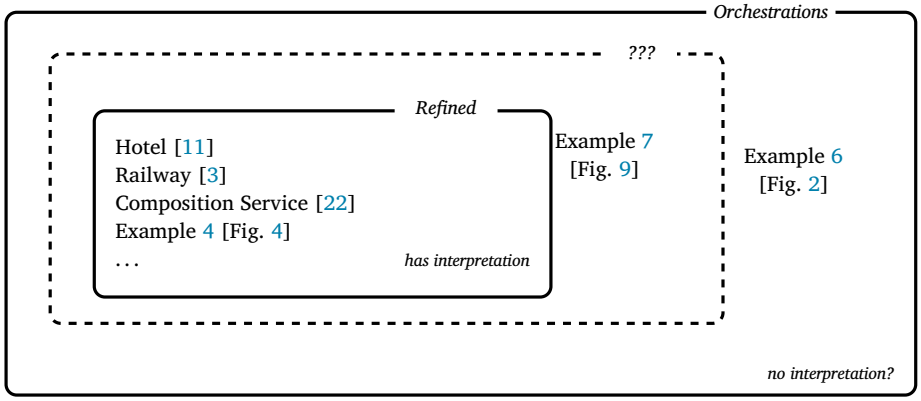


Fig. 5. A Venn diagram showing the set of orchestrations of contract automata.

then t is *controllable* in $\mathcal{A}'$ (via t'). Otherwise, t is *uncontrollable* in $\mathcal{A}'$.

By comparing Definition 2 and Definition 5, we note that only the semi-controllable transitions have been refined, while the others remain unaltered. Conditions 1 and 2 of Definition 5 contain the constraints that are used to decide when a semi-controllable transition is controllable or uncontrollable. The constraints of Condition 1 are also present in Definition 2. The intuition is that a (refined) semi-controllable transition t becomes controllable if (similarly to Definition 2) in a given portion of $\mathcal{A}$, there exists a semi-controllable match transition t', with source and target states not dangling, such that in both t and t' the *same* service, in the *same* local state, does the *same* request. Condition 2 of Definition 5 imposes new further constraints. It requires that t' is *reachable* from the source state of t through a sequence of transitions where the service performing the request is idle.

Consider the Venn diagram in Fig. 5. The outermost set *Orchestrations* contains all orchestrations of contract automata that are computed using the notion of semi-controllability of Definition 2. The innermost set *Refined* contains only those orchestrations that are computed using the refined notion of semi-controllability in Definition 5. Intuitively, the refined notion imposes a further constraint on *when* a semi-controllable transition is controllable. As a result, more semi-controllable transitions are uncontrollable than in the previous definition. This explains why *Refined* is contained in *Orchestrations*.

All the examples of semi-controllability available in the literature [34,11,2,22] (e.g., Hotel service) and in Fig. 4 are orchestrations that belong to the set *Refined* in Fig. 5. This means that by updating the notion of semi-controllability, all orchestrations of these examples remain unaltered.

Example 6. We now provide an example of an orchestration belonging to *Orchestrations* $\setminus$ *Refined* (cf. Fig. 5). We have three principal contracts, namely **A**lice, **B**ob and **C**arl, depicted in Fig. 1. The contracts of **B**ob and **C**arl perform two alternative necessary requests. The contract of **A**lice has two branches. In each branch, a request of **B**ob and a request of **C**arl are fulfilled by corresponding offers.

Using the notion of semi-controllability from Definition 2, the synthesis algorithm of Definition 3 takes as input the composed automaton and returns the (conditional) orchestration of the composition, depicted in Fig. 2, which is a contract of rank 3. Indeed, for each necessary request of each service, there *exists* a match transition in the composition where the necessary request is fulfilled by a corresponding offer. In other words, for each necessary request of **B**ob and **C**arl, there exists an execution where the request is matched by a corresponding offer. For example, the composition **A**lice $\otimes$ **B**ob $\otimes$ **C**arl contains the transition $t = [\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0] \xrightarrow{[-,?d,-]_\square} [\mathrm{a}_1, \mathrm{b}_2, \mathrm{c}_0]$, which is semi-controllable. According to Definition 2, t is controllable (in **A**lice $\otimes$ **B**ob $\otimes$ **C**arl) via $t' = [\mathrm{a}_2, \mathrm{b}_0, \mathrm{c}_0] \xrightarrow{[!d,?d,-]_\square} [\mathrm{a}_4, \mathrm{b}_2, \mathrm{c}_0]$. Since t is controllable and it is not in agreement (i.e., the label of t is a request), this transition is pruned during the synthesis of the orchestration. We note that t is controllable in t' also in all the sub-automata of the composition computed in the various iterations of the synthesis algorithm, and in the final orchestration depicted in Fig. 2.

Using the refined notion of semi-controllability of Definition 5, the orchestration of **A**lice $\otimes$ **B**ob $\otimes$ **C**arl is empty (i.e., there is no orchestration). Consider again transition t. From state $[\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0]$, it is not possible to reach any transition labelled by $[!d, ?d, -]_\square$. It follows that t is uncontrollable. Hence, at some iteration i of the (conditional) orchestration synthesis algorithm in Definition 3, state $[\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0]$ becomes forbidden and it is added to the set R_i. At iteration $i+1$, the controllable transition $[\mathrm{a}_0, \mathrm{b}_0, \mathrm{c}_0] \xrightarrow{[!a,-,-]_\square} [\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0]$ is pruned because its target state is forbidden. At the next iteration $(i+2)$, the initial state $[\mathrm{a}_0, \mathrm{b}_0, \mathrm{c}_0]$ becomes forbidden, because there are semi-controllable transitions not in agreement exiting the initial state (e.g., $[\mathrm{a}_0, \mathrm{b}_0, \mathrm{c}_0] \xrightarrow{[-,?c,-]_\square} [\mathrm{a}_0, \mathrm{b}_1, \mathrm{c}_0]$) that are uncontrollable in the sub-automaton whose transitions are T_{i+2}. Since the initial state is forbidden, it follows that there is no orchestration for **A**lice $\otimes$ **B**ob $\otimes$ **C**arl.

Indeed, whenever the state $[\mathrm{a}_1, \mathrm{b}_0, \mathrm{c}_0]$ is reached, although **B**ob and **C**arl are still in their initial state, **B**ob can no longer perform the necessary request $?d$ and **C**arl can no longer perform the request $?f$. In fact, neither **B**ob nor **C**arl can decide internally which necessary request to execute from their current state. For example, there is no trace where the request $?c$ of **B**ob and the request $?f$ of **C**arl are matched.

We now give the intended intuitive interpretation of semi-controllability and compare it to the classical notion of uncontrollability. We recall that uncontrollable transitions are called *urgent* necessary transitions in MSCA, while semi-controllable transitions are called *lazy* necessary transitions. Recall that, intuitively, an urgent transition cannot be delayed, whereas this is the case for a lazy one. In a concurrent composition of agents, the scheduling of concurrent urgent necessary transitions is *uncontrollable*. Instead, concerning concurrent lazy necessary transitions, each agent *internally* decides its next lazy necessary transition to execute, but the orchestrator schedules when this transition will be executed, i.e., the scheduling is *controllable*. In Example 6, there is no (conditional) orchestration because, for example, from state $[a_1, b_0, c_0]$ there is no possible scheduling that allows the services to match all their necessary requests. Continuing Example 4, the orchestration in Fig. 4 is non-empty because the scheduling of the actions in the orchestration is *controlled* by the orchestrator: one of the two necessary requests is scheduled to be matched only when the server has reached its internal state [2]. If instead the clients' necessary request $?a$ is urgent, then there exists no orchestration of the composition of two clients and the server. This is because in this case the scheduling is *uncontrollable*: it is not possible to schedule one of the two clients to have its necessary urgent request to be matched only when the server reaches the state [2]. In this case, the server should be ready to match the requests whenever they can be executed, without delaying them.

4. Research questions

In this section, we describe the currently known limits of the synthesis of orchestrations adopting either Definition 2 or Definition 5, and we identify a number of research questions to overcome these limits, which will be addressed in Section 5.

First, the notion of semi-controllability introduced in [11,12] and recalled in Definition 2 allows to synthesise orchestrations that may sometimes limit the capability of each service to perform internal choices. Originally, the contract automata formalism was abstracting from the way that choices are made. Different implementations are possible in which each service may or may not decide the next step in an orchestration [22].

Consider again Example 6. Both **B**ob and **C**arl are able to perform two alternative necessary requests from their initial state. However, as shown in Fig. 2, they are forbidden from internally deciding which necessary request is to be executed at runtime. If, for example, **B**ob selects the request $?d$ and **C**arl selects the request $?e$, then it is not possible for **A**lice to match both requests. In other words, the orchestrator not only controls the scheduling but also controls the choices made by each service contract.

If we adopt the interpretation given previously (i.e., agents internally choose their necessary transitions and their scheduling is controllable) then we argue that the (conditional) orchestration computed using Definition 2 is too abstract (i.e., we abstracted away whether choices are internal or external) and should in fact be empty. This is indeed the case if Definition 5 were used instead of Definition 2.

RQ1: The first research question is to identify a concrete application of services that perform necessary requests and whose orchestration belongs to the set $\mathit{Orchestrations} \setminus \mathit{Refined}$.

Solving this research question could help provide an intuitive interpretation of these types of orchestrations. An application should be identified in which each service statically requires that for each necessary request there must exist an execution where this is eventually matched (cf. Definition 2). However, during execution, the choice of which necessary request is to be matched could be *external* to the service performing the necessary request. Even if the execution of different branches is determined externally, a service contract may still require all branches to be available in the composition. This could be due to the contract's need to enforce certain hyperproperties, such as non-interference or opacity [35].

Next, we illustrate the second research question. All examples of orchestrations currently available in the literature [3,22,12,2,11] reside inside the set *Refined* (cf. Fig. 5). We showed in Example 6 an orchestration **O** not belonging to the set *Refined* and we argued that **O** is too abstract and should in fact be empty. We now provide another example of an orchestration not belonging to the set *Refined*. However, differently from Example 6, in this case the orchestration should not be empty.

Example 7. This example involves a simple card game with two players and a dealer. At the beginning of each round, the dealer chooses a pair of cards to deal to each player (i.e., each player receives two cards). The dealer can select two out of three different pairs of cards:

- Pair 1: card 1 and card 3;
- Pair 2: card 2 and card 4;
- Pair 3: card 2 and card 3.

After the dealer has dealt the pairs of cards, each player selects one of the two cards that was received. Once the players have selected their cards, the dealer collects the selected cards from each player. The goal of the game is for the dealer to avoid picking up two cards in equal or in ascending order, which would result in the dealer losing. In other words, if the dealer picks up a card that is higher than the other card that was picked up or if two cards of the same value are picked up, the dealer loses. To ensure that the dealer never loses, the dealer has to choose the correct pairs of cards to deal. There are six possible ways to choose the pairs of cards, but only two of them guarantee a strategy for the dealer to collect the cards selected by the players in descending order. The strategy

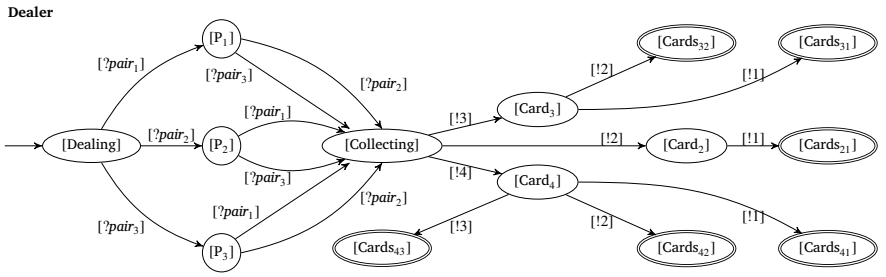


Fig. 6. Contract of the **Dealer**.

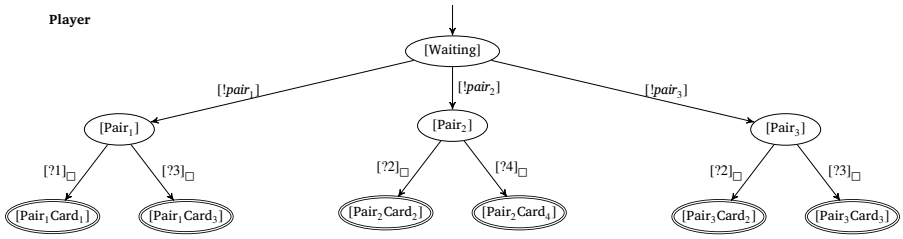


Fig. 7. Contract of the **Player**.

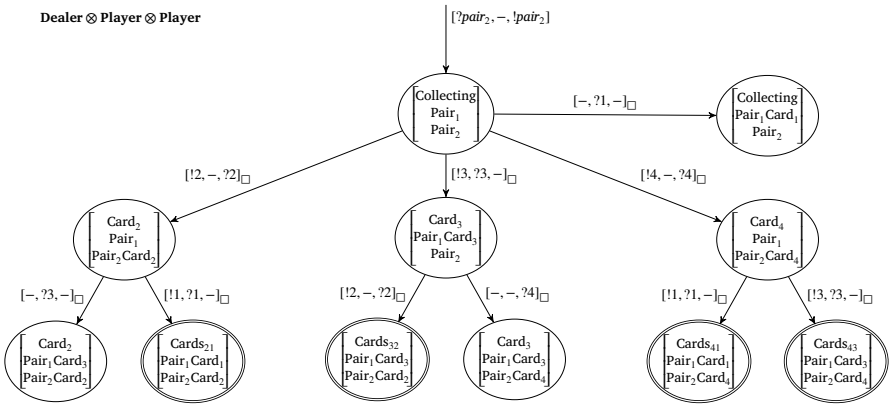


Fig. 8. A fragment of the composition of **Dealer** ⊗ **Player** ⊗ **Player**.

for the dealer consists of dealing to the players (in no particular order) Pair 1 and Pair 2. Indeed, in the remaining cases there exists the possibility that the players *internally* select the same card. In this case, there is no way of rearranging the transitions to avoid the same cards being picked by the dealer.

We modelled this above mentioned problem as an orchestration of contracts, using the refined notion of semi-controllability. We only model one round of the game. The CA in Fig. 6 models the dealer. Note that each request can be matched by either of the two players. Once the dealer has dealt the pairs of cards, the cards selected by the players are collected. Note that the two cards can only be collected in descending order. The CA in Fig. 7 models a player. Once the player has received a card, the player decides internally which card to select. This internal decision is modelled as a choice among lazy necessary transitions.

The synthesis algorithm adopting the refined notion of semi-controllability from Definition 5 takes as input the composition of three CA, namely the dealer and two players, and returns an empty orchestration. To explain why the resulting orchestration is empty, consider Fig. 8 depicting a portion of the composition of the dealer with two players.

The state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ is reached when the first player receives $pair_1$ and the second player receives $pair_2$. A symmetric argument holds for state $[\text{Collecting}, \text{Pair}_2, \text{Pair}_1]$, not depicted here. The transition

$$[\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2] \xrightarrow{[-,?3,-]_\square} [\text{Card}_2, \text{Pair}_1\text{Card}_3, \text{Pair}_2\text{Card}_2]$$

is uncontrollable according to Definition 5. Indeed, it is not possible to reach state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ from state $[\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$. This makes the state $[\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$ forbidden. Hence, to avoid reaching a forbidden state, the algorithm prunes the transition

$$[\text{Collecting}, \text{Pair}_1, \text{Pair}_2] \xrightarrow{[!2,-,?2]_\square} [\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$$

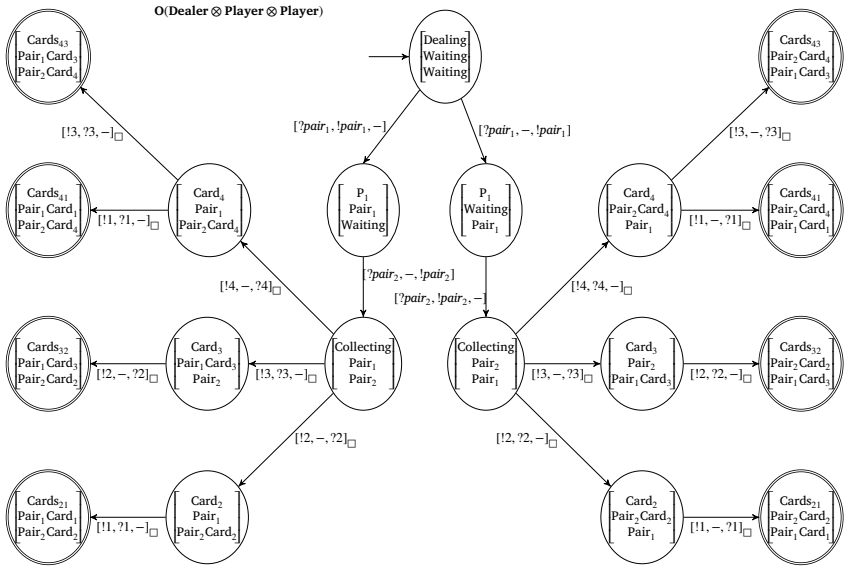


Fig. 9. Orchestration **O(Dealer ⊗ Player ⊗ Player)**.

which is in fact controllable according to Definition 5. Indeed, from state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ it is possible to reach the transition

$$[\text{Card}_3, \text{Pair}_1\text{Card}_3, \text{Pair}_2] \xrightarrow{[!2,-,?2]_\square} [\text{Cards}_{32}, \text{Pair}_1\text{Card}_3, \text{Pair}_2\text{Card}_2]$$

via a transition in which the second player is idle. However, during the synthesis algorithm also the state $[\text{Card}_3, \text{Pair}_1\text{Card}_3, \text{Pair}_2]$ becomes forbidden due to its outgoing necessary transition, which is uncontrollable according to Definition 2. This in turn causes the pruning of transition

$$[\text{Collecting}, \text{Pair}_1, \text{Pair}_2] \xrightarrow{[!3,?3,-]_\square} [\text{Card}_3, \text{Pair}_1\text{Card}_3, \text{Pair}_2]$$

which is controllable. Once the transition has been pruned, the transition

$$[\text{Collecting}, \text{Pair}_1, \text{Pair}_2] \xrightarrow{[!2,-,?2]_\square} [\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$$

which was previously controllable now becomes uncontrollable (recall that this can only happen if the transition has been pruned during a previous iteration). This makes the state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ forbidden. Note, however, that the state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ should *not* be forbidden. Indeed, from that state, for each pair of cards selected by the players, the dealer has a strategy to pick them in the correct order:

- if player 1 selects card 1 and player 2 selects card 2, then execute $[!2, -, ?2], [!1, ?1, -]$;
- if player 1 selects card 1 and player 2 selects card 4, then execute $[!4, -, ?4], [!1, ?1, -]$;
- if player 1 selects card 3 and player 2 selects card 2, then execute $[!3, ?3, -], [!2, -, ?2]$;
- if player 1 selects card 3 and player 2 selects card 4, then execute $[!4, -, ?4], [!3, ?3, -]$.

This example shows that there are cases for which Definition 5 is too restrictive. In this case, an orchestration exists and it is displayed in Fig. 9. In Section 5, we will show how to compute this orchestration.

To better understand the underlying assumption of Definition 5, we need to decouple the moment in which a service *selects* which transition it will execute from the moment in which a service *executes* that transition. The underlying assumption of Definition 5 is that these two moments are not decoupled. For example, the first player whose internal state is Pair_1 could select and execute ?3 also from state $[\text{Card}_2, \text{Pair}_1, \text{Pair}_2\text{Card}_2]$, while the strategy described above assumes that the player selects a card in state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$.

In fact, the current implementation of the contract automata runtime environment CARE [22] allows the decoupling of these two moments. Once state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ is reached, the orchestrator interacts with both players and, based on their choices, correctly schedules the transitions of the dealer and the players. This means that the players select their next action in state $[\text{Collecting}, \text{Pair}_1, \text{Pair}_2]$ and afterwards their execution is bounded to the transition they have selected. Summarising, Example 6 has showed that in some cases Definition 2 is too abstract (i.e., a non-empty orchestration is computed although there is no orchestration), whereas Example 7 has showed that in some cases Definition 5 is too restrictive (i.e., an empty orchestration is computed although a non-empty orchestration exists).

RQ2: The second research question is to identify a notion of semi-controllability capable of discarding orchestrations such as the one in Example 6 and providing non-empty orchestrations in scenarios such as the one described in Example 7.

The resulting set of orchestrations that would be identified by the notion of semi-controllability that solves this research question is depicted in Fig. 5 with dashed lines.

We continue by discussing further research questions for the orchestration synthesis of contract automata. An important aspect is the ability to scale to large orchestrations when many service contracts are composed. We note that computing Definition 5 is harder than computing Definition 2, due to the additional constraint of reachability which requires a visit of the automaton. Decoupling the moment in which a service selects a choice from the moment in which the selected choice is executed, could further increase the 'hardness' of deciding when a lazy necessary transition is controllable.

Consider again the CA in Fig. 1. From their initial state, both **B**ob and **C**arl have two choices. If, instead of two principals, we had ten principals whose behaviour is similar to that of **B**ob and **C**arl, then there would be 2^{10} possible combinations of (internal) choices the services could make.

RQ3: The third research question is to study scalable solutions for synthesising orchestrations.

Generally speaking, the behaviour of an orchestration that belongs to the unknown dashed set of Fig. 5 must be a sub-automaton of an orchestration computed using Definition 2 and a super-automaton of an orchestration computed using Definition 5. Indeed, Definition 2 can be used as an upper bound and Definition 5 as a lower bound to approximate the behaviour of such an orchestration.

Finally, we discuss the last research question we identify in this paper. We previously formalised the notion of lazy necessary request that is semi-controllable according to either Definition 2 or Definition 5. We noted that Definition 2 may exclude the case in which, in the presence of a choice, a service may internally select its necessary transition. Instead, Definition 5 may exclude the case in which, in the presence of a choice, the moment in which the service internally selects its necessary transition is decoupled from the moment in which the selected necessary transition is executed. In other words, we identified two *requirements* that an orchestration of services should satisfy: *independence* and *decoupling* of choices.

RQ4: The fourth research question is to consolidate a set of requirements that a desirable orchestration of service contracts must satisfy.

The requirements that would solve this research question should be established incrementally, as discussed in this paper. Formal definitions of necessary service transitions and practical examples are useful to identify the ideal set of requirements that an orchestration of services should satisfy. Of course, these requirements are entangled with the underlying execution support of an orchestration of services, which was recently proposed in [22].

5. Advancing orchestration synthesis for contract automata

In the previous section, we presented a series of four research questions associated with the orchestration of contract automata. In this section, we address each of them, thus advancing the orchestration synthesis for contract automata.

5.1. First research question: specifying choices

A simple solution to the first research question is to establish the interpretation of necessary requests introduced previously. Under this interpretation, a service *internally* decides to perform a necessary request, but the scheduling of the execution of the request is controlled by the orchestrator. In other words, optional actions are externally selected, whereas necessary actions are internally selected. This implies that currently no concrete application of services performing externally selected necessary transitions has been identified.

Until now, the distinction between internal and external selections was abstracted away within contracts and handled by the underlying execution support. Abstracting from the selection process leads to scalability issues. Specifically, if a transition is selected internally, it must always be available, whereas an externally selected transition can be removed from the orchestration. In essence, internal selection imposes stricter requirements than external selection. By modelling internal and external selections at the contract automata level, through necessary and optional actions, respectively, we improve scalability. For example, the issue highlighted in Example 6 arises due to the presence of externally selected transitions. By allowing contracts to specify which transitions are selected internally and which externally, we can reduce the state space, as compared to considering all choices as internal.

RQ1 Summary: By assuming that all necessary requests are internally selected, scenarios like the one outlined in the context of the first research question (i.e., external necessary requests) are practically ruled out.

Remark Note that the initial definition of semi-controllability (i.e., Definition 2) permits the modelling of externally selected necessary transitions. To address the first research question, we eliminate these specific necessary transitions. Specifically, we enforce that necessary transitions of contract automata are exclusively internally selected. The updated orchestration synthesis, with necessary transitions internally selected, will be discussed in the next section. While an alternative approach could involve allowing to express both internally and externally selected necessary transitions, we opted for removing the notion of externally selected transition. This decision is based on the absence of practical examples illustrating such constructs at present. Despite this modification, Definition 2 remains useful as an overapproximation, as we discuss later.

5.2. Second research question: decoupling of choices

The second research question, as mentioned previously, revolves around the absence of the decoupling of choice requirement in both Definitions 2 and 5. The decoupling of choice in fact suggests as solution a new notion of semi-controllable transition. This novel notion identifies the currently unknown (dashed) set of orchestrations in Fig. 5.

Currently, a semi-controllable transition is defined as a transition that can be either controllable or uncontrollable based on a global condition of the automaton. However, decoupling the moment in which a service internally selects a transition from the moment in which the transition is executed, in fact requires splitting a semi-controllable transition into two distinct transitions.

Reasoning in this way suggests that a semi-controllable transition can be represented as two consecutive transitions. The first transition is uncontrollable, capturing the internal selection, while the subsequent transition is controllable and responsible for executing the action. For example, consider a semi-controllable transition $[\mathtt{q}] \xrightarrow{[?\mathtt{a}]_{\square_l}} [\mathtt{q}']$, which will be split into two transitions: $t_1 = [\mathtt{q}] \xrightarrow{[\tau_a]_{\square_u}} [\mathtt{i}]$ and $t_2 = [\mathtt{i}] \xrightarrow{[?\mathtt{a}]} [\mathtt{q}']$. Here, t_1 represents an uncontrollable silent transition to an intermediate, non-final state (`[i]`), while t_2 is controllable and executes the action (`[?a]`). With this approach, the orchestrator cannot control the internal selection made with t_1, but it can control and schedule the execution of the action indicated by t_2. Moreover, an important consequence of the fact that the intermediate state is non-final, is that t_2 must eventually be executed.

This new notion of semi-controllability affects contract automata and their orchestration synthesis algorithm. Indeed, in a contract automaton all necessary requests are categorised as lazy/semi-controllable, thus effectively excluding urgent necessary requests from contracts. Contract automata with optional and necessary transitions are subsequently transformed into automata with solely optional/controllable and urgent/uncontrollable transitions, which are known as plant automata in supervisory control theory [4,5]. All uncontrollable transitions will serve as silent moves to represent the internal selection of a necessary transition. Since the encoded automata only have controllable and uncontrollable transitions, the standard mpc synthesis (cf. Definition 4) can subsequently be applied to the encoded automata, to synthesise the orchestration.

Expressing internal actions Currently, the basic actions of a contract automaton are offers, requests and a distinguished idle action. Since the encoded urgent transition models an internal selection, it must be labelled with an internal τ action. We extended contract automata to be able to express τ actions. These are special actions that are ignored by the properties of agreement and strong agreement, and cannot be matched by any contract. For example, a contract automaton with solely match transitions and τ transitions satisfies both agreement and strong agreement.

We note that τ actions are syntactic sugar that can be expressed within the existing contract automata formalism. For example, the necessary transition $[\mathtt{q}] \xrightarrow{[?\mathtt{a}]_{\square_l}} [\mathtt{q}']$ can straightforwardly be encoded using an additional artificial service, which matches the transition that encodes the internal selection, i.e., $t_1 = [\mathtt{q}, \mathtt{q}_\tau] \xrightarrow{[?\mathtt{tau\_a}, !\mathtt{tau\_a}]_{\square_l}} [\mathtt{i}, \mathtt{q}_\tau]$ and $t_2 = [\mathtt{i}, \mathtt{q}_\tau] \xrightarrow{[?\mathtt{a}, -]} [\mathtt{q}', \mathtt{q}_\tau]$. By the definition of composition of contract automata $\otimes$, all other services that will be added later to the composition cannot interfere with the internal match $[?\mathtt{tau\_a}, !\mathtt{tau\_a}]_{\square_l}$, which acts effectively as an internal transition. Nevertheless, to simplify the encoding, we opted for including τ as a new type of action that can be expressed within contract automata. Thus, Definition 1 is updated as follows. The set of internal actions A^τ is added to the tuple forming an MSCA (i.e., the tuple becomes $\langle Q, \vec{q}_0, A^r, A^o, A^\tau, T^{\square_u}, T^{\square_l}, T^{\bigcirc}, F\rangle$). As a side effect, the set A is now a subset of $(A^r \cup A^o \cup A^\tau \cup \{-\})^n$.

5.2.1. Splitting orchestration synthesis

We now define the new orchestration synthesis algorithm that exploits the novel notion of semi-controllability we have just described above. We refer to this new orchestration synthesis as *splitting* orchestration synthesis, because each semi-controllable transition is split into two transitions, one uncontrollable and one controllable. Definition 6 formalises the splitting orchestration synthesis and the new notion of semi-controllability. The splitting orchestration synthesis takes as input the list of principal contract automata to be orchestrated (i.e., $rank(\mathcal{A}_i) = 1$). Alternatively, the synthesis could take as input a composed automaton and extract all principals contract automata using the projection operator of contract automata [10].

As stated in Section 2, the orchestration synthesis (both conditional and splitting) only considers necessary transitions that are lazy/semi-controllable. Each principal contract $\mathcal{A}_i$ is encoded into a new principal contract $\mathcal{A}_i^e$, implementing the encoding of lazy/semi-controllable request transitions into pairs of concatenated urgent/uncontrollable internal transitions and optional/controllable request transitions. In $\mathcal{A}_i^e$, each semi-controllable lazy transition $t = (\vec{q}, \vec{a}, \vec{q}')_{\square_l}$ is substituted by two transitions $(\vec{q}, \vec{\tau}_a, \vec{q}_t)_{\square_u}$ and $(\vec{q}_t, \vec{a}, \vec{q}')_{\bigcirc}$, where q_t is a fresh intermediate state, and $\vec{\tau}_a$ is a vector of rank 1 where the internal action modelling the choice of action a is performed. The obtained encoded principals are such that no lazy/semi-controllable transitions are present, and all urgent/uncontrollable transitions are internal actions. It is then possible to apply the standard mpc synthesis of Definition 4 on the composition of the encoded contracts.

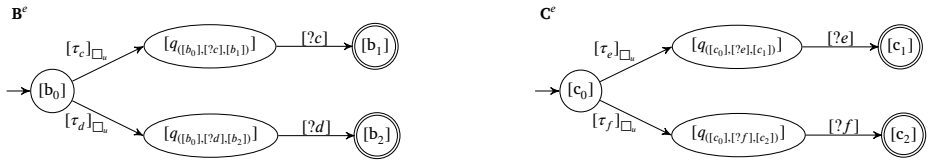


Fig. 10. Contracts of $\mathbf{Bob}^e$ and $\mathbf{Carl}^e$.

Definition 6 *(Splitting orchestration synthesis).* Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be principal MSCA such that $\forall i \in 1 \ldots n$, we have $T^{\Box_u}_{\mathcal{A}_i} = \varnothing, A^{\tau}_{\mathcal{A}_i} = \varnothing$, and $\mathcal{A}^e_i = \langle Q_{\mathcal{A}_i} \cup \{\, [q_t] \mid t \in T^{\Box_l}_{\mathcal{A}_i} \,\}, \vec{q}_{0\mathcal{A}_i}, A^r_{\mathcal{A}_i}, A^o_{\mathcal{A}_i}, A^{\tau}, T^{\Box_u}, T^{\Box_l}, T^{\bigcirc}, F_{\mathcal{A}_i} \rangle$, where:

- $A^{\tau} = \{\, \tau_a \mid (\vec{q}, [?a], \vec{q}') \in T^{\Box_l}_{\mathcal{A}_i} \,\}$;
- $T^{\Box_u} = \{\, (\vec{q}, [\tau_a], [q_t]) \mid t = (\vec{q}, [?a], \vec{q}') \in T^{\Box_l}_{\mathcal{A}_i} \,\}$;
- $T^{\Box_l} = \varnothing$;
- $T^{\bigcirc} = T^{\bigcirc}_{\mathcal{A}_i} \cup \{\, ([q_t], [?a], \vec{q}') \mid t = (\vec{q}, [?a], \vec{q}') \in T^{\Box_l}_{\mathcal{A}_i} \,\}$.

Then $\mathcal{K}_{\mathcal{A}^e_1 \otimes \ldots \otimes \mathcal{A}^e_n}$ is the orchestration of the principals $\mathcal{A}_1, \ldots, \mathcal{A}_n$.

Note that the splitting orchestration in Definition 6 is different from the conditional orchestration discussed in Definition 3. Indeed, while Definition 3 is a variant of Definition 4, Definition 6 is an extension that reuses Definition 4. This is an important aspect because the splitting orchestration can thus be implemented not only in `CATLib` [12], which supports the mpc synthesis, but also in any other tool implementing the mpc synthesis.

By adopting Definition 6, the orchestration of Example 6 is empty, whereas the orchestration of Example 7 is non-empty.

Example 8. We continue Example 6. By adopting Definition 6, the orchestration of $\mathbf{A} \otimes \mathbf{B} \otimes \mathbf{C}$ is empty. Fig. 10 shows the automata $\mathbf{B}^e$ and $\mathbf{C}^e$, which are the encoded versions (Definition 6) of $\mathbf{B}$ and $\mathbf{C}$ from Fig. 1. The principal contract $\mathbf{A}$ has no lazy requests, thus $\mathbf{A} = \mathbf{A}^e$.

Consider, for example, the lazy/semi-controllable transition $([b_0, ?c, b_1])_{\Box_l}$ of $\mathbf{B}$ in Fig. 1. In $\mathbf{B}^e$, this transition unfolds in two transitions. The first one is $([b_0], [\tau_c], [q_{([b_0],[?c],[b_1])}])_{\Box_u}$. This transition models the internal selection of the transition $([b_0], [?c], [b_1])_{\Box_l}$ of $\mathbf{B}$. The internal selection is in fact rendered as an urgent/uncontrollable transition labelled by a τ action. The second transition is $([q_{([b_0],[?c],[b_1])}], [?c], [b_1])$. This transition is optional/controllable, and models the execution of the transition $([b_0], [?c], [b_1])_{\Box_l}$ of $\mathbf{B}$. Through this encoding, the moments of selection and execution of a lazy/semi-controllable transition are decoupled. Furthermore, the intermediate state $[q_{([b_0],[?c],[b_1])}]$ is non-final. Thus, in order to fulfil the contract of $\mathbf{B}$, the only optional transition leaving this intermediate state must eventually be executed.

It is easy to see that the resulting orchestration is empty. For example, if we consider the trace

$$[a_0, b_0, c_0] \xrightarrow{[-,\tau_c,-]_{\Box_u}} [a_0, q_{([b_0],[?c],[b_1])}, c_0] \xrightarrow{[-,-,\tau_f]_{\Box_u}} [a_0, q_{([b_0],[?c],[b_1])}, q_{([c_0],[?f],[c_2])}]$$

then the orchestrator cannot prevent the execution of these two transitions, since they are uncontrollable and lead to a non-final intermediate state. From this intermediate state, it is only possible to reach a final state by matching the requests $?c$ and $?f$ from $\mathbf{B}^e$ and $\mathbf{C}^e$, respectively. However, $\mathbf{A}^e$ can only match one of the two requests, but not the other. Consequently, during the mpc synthesis (cf. Definition 4), the above mentioned intermediate state will be marked as forbidden (i.e., belonging to the set R). Eventually, also the initial state will be marked as forbidden, thus returning an empty orchestration.

Example 9. Continuing Example 7, the (splitting) orchestration of the card game computed with Definition 6 is non-empty. One side of the orchestration (i.e., the first player receives `pair1` and the second player receives `pair2`) is depicted in Fig. 11. The symmetric case is omitted for displaying purposes.

RQ2 Summary: The splitting orchestration synthesis in Definition 6 solves the second research question, and identifies the unknown set of orchestrations in Fig. 5.

Implementation The conditional orchestration synthesis from Definition 3, using the semi-controllability of Definition 2, is implemented in `CATLib` [12], under the class `OrchestrationSynthesisOperator.java`. This class has been updated to implement Definition 5 as follows. A Boolean flag called `refinedLazy` has been added to the class. When this flag is set, the refined semi-controllability in Definition 5 is used, i.e., in addition to the conditions formalised in Definition 2, also the further reachability conditions formalised in Definition 5 are checked. Otherwise, the original semi-controllability of Definition 2 is used.

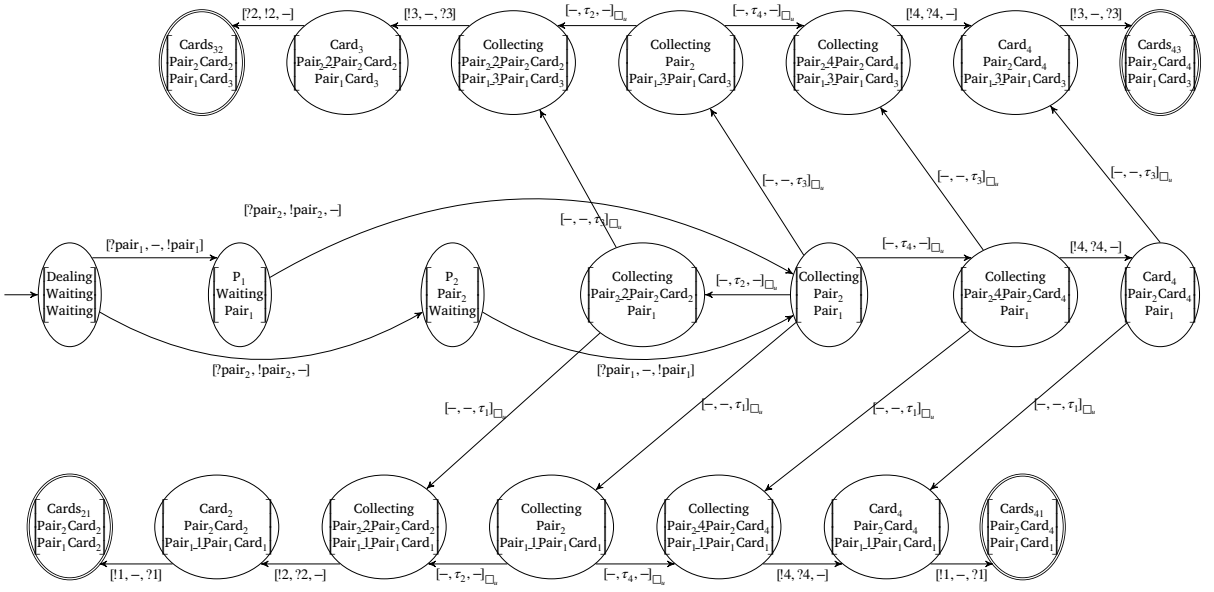


Fig. 11. One side of the orchestration of Example 9.

Concerning the splitting orchestration synthesis from Definition 6, this has been implemented under the `SplittingOrchestrationSynthesis.java` class, which extends the already existing class `MpcSynthesisOperator.java`. Basically, the class implements the encoding and invokes the synthesis method of its super class on the composition of the encoded automata. Alternatively, the class offers a method taking as input an automaton of rank greater than one. Before calling the orchestration synthesis method, the list of principal contracts is extracted from the composition.

`CATLib` has also been updated to include a new type of action, implemented in the new class `TauAction.java`. The source code has been updated accordingly to deal with this new type of action.

The new implementation is available in a permanent repository at [14], and has been exploited to perform different experiments, which we discuss next.

5.2.2. *Third research question: scalability*

The third research question highlights the issue of scalability. In this subsection, we discuss the latest improvements of `CATLib` towards more efficient orchestration syntheses. The source code of the experiments performed in this section is open source and publicly available [14].

Committed states Contract automata have been equipped with committed states to reduce the state space of a composition and to improve expressivity. Whenever a state $\vec{q}$ has a committed element $\vec{q}_{(i)}$, then all transitions outgoing that state must have a label $\vec{a}$ such that $\vec{a}_{(i)} \neq -$. In other words, whenever two concatenated transitions of a service have the intermediate state (i.e., target state of one transition and source state of the other transition) committed, then the two transitions are executed atomically: after the service has executed the first transition, the second transition is executed prior to any other transition of any other service in the composition. In this way, the number of interleavings of transitions and states in a composition is reduced. Furthermore, contract automata primitively support one-to-one synchronisations (called match transitions). Through committed states it is also possible to express one-to-many and many-to-one synchronisations (as is common in UML or in team automata [36,29]).

Example 10. Continuing Example 7, we can now reveal that the transitions of the dealer (Fig. 6) labelled by $?pair_1, ?pair_2$ and $?pair_3$ are executed atomically. In other words, both players are allowed to select their card only after both have received their pair. Otherwise it would be possible in the composition $\mathcal{A} = \textbf{Dealer} \otimes \textbf{Player} \otimes \textbf{Player}$ for a player to select a card before the other player has received its pair of cards. In this case, a different strategy would be enabled where the dealer first provides $pair_3$ to one of the two players, and afterwards decides the pair to give to the other player based on which card has been selected by the player with $pair_3$. Note that in this example, we target the expressiveness of contract automata (i.e., the ability to express transitions executed atomically), and as a side effect we improve scalability by reducing the state space.

Within `CATLib`, it is possible to ensure that the pairs are provided to both players before they select the cards in two alternative ways, either by model checking or by using committed states.

In the former case, it is possible to exploit the model checking capability of `CATLib` [12], which allows to model check a contract automaton against a property expressed as a finite state automaton. In `CATLib`, the composition of contract automata and the model checking function are both specialisations of the same class `CompositionFunction.java`. Basically, the model checking function is the synchronous product of a contract automaton with a simple finite state automaton expressing a property (ignoring whether the

Table 1
Comparison of different state space reduction techniques for the card game.

	Time	*States*	*Transitions*
Plain Composition	337 ms	560	935
Composition + Committed States	137 ms	177	224
Composition + Pruning	52 ms	85	94
Composition + Pruning + Committed States	82 ms	73	82
Composition + Pruning + Committed States + Symmetric Reduction	126 ms	36	37

actions of the contract are requests, offers or matches). The composition $\mathcal{A}$ is firstly model checked against the property (expressed here as a regular expression)

$$P = (pair_1 \mid pair_2 \mid pair_3)(pair_1 \mid pair_2 \mid pair_3)(1 \mid 2 \mid 3 \mid 4)^*$$

The property P only allows sequences of actions in $\mathcal{A}$ in which cards are selected only after the two pairs have been provided. The model checking of $\mathcal{A}$ against P produces an automaton $\mathcal{A}'$ containing the traces of $\mathcal{A}$ that are also in P. Afterwards, the orchestration synthesis is applied to $\mathcal{A}'$.

Note that the model checking problem $\mathcal{A} \vDash P$ can be decided by turning all transitions of $\mathcal{A}$ to uncontrollable and checking the non-emptiness of $\mathcal{A}'$ (hence the name model checking for this composition).

Before applying the model checking function, it is possible to instruct `CompositionFunction.java` to ignore the modalities of $\mathcal{A}$. In this example, the transitions of $\mathcal{A}$ violating P are not present in $\mathcal{A}'$, even if they are necessary, since modalities are ignored by the model checking function (otherwise it would not be possible to block a player from performing a necessary request (e.g., $[?2]_{\square}$) before both players have received their pair). Basically, in this example the requirement imposed by P has precedence over the modalities imposed by the services over their requests. Finally, in this first case, the composition $\mathcal{A}$ must be first generated, and afterwards the transitions and states of $\mathcal{A}$ not satisfying P are removed in $\mathcal{A}'$.

In the second case, the states $[P1], [P2]$ and $[P3]$ of the dealer in Fig. 6 are modelled as committed. In this way, the property P is expressed directly by the dealer automaton, whose transitions providing the pairs of cards to both players are executed atomically. More importantly, the states and transitions violating P are never generated.

Table 1 shows a comparison of the size of the composed automaton $\mathcal{A}$ using different techniques for reducing the state space, and the time needed for computing them. The plain composition has only the original constraint [10] preventing two services from interleaving their request and offer transitions whenever the corresponding match transition is enabled. Committed states are not used in the plain composition (in this case, property P will be enforced afterwards by performing the model checking composition of $\mathcal{A}$ with P). This is the largest composition, having 560 states and 935 transitions. By only including the committed states of the dealer automaton in the plain composition, the resulting composition has 177 states and 224 transitions.

As visible in Fig. 9, in the orchestration we are only interested in match transitions. The property allowing only match transitions is called *strong agreement* (no request or offer transitions are allowed). During the state space generation, the composition of `CATLib` takes as parameter a *pruning predicate*. The pruning predicate takes as input a transition and returns the Boolean value *true* if the transition is violating the property we want to enforce. For example, if an optional transition is not a match, then the pruning predicate enforcing strong agreement will evaluate to *true* (note that a necessary transition cannot be pruned at composition time). When computing the composition, the pruning predicate is used to avoid generating redundant portions of the state space that will anyway be removed by the orchestration synthesis. The composition with the pruning predicate (enforcing strong agreement) and without committed states has 85 states and 94 transitions, whereas the one with committed states has 73 states and 82 transitions.

Finally, we note in Fig. 9 that the state space can be halved. Indeed, the case in which the first player receives $pair_1$ and the second player receives $pair_2$ is symmetrical to the case where the first player receives $pair_2$ and the second player receives $pair_1$. Therefore, the dealer can be instructed to check only the three combinations $(pair_1, pair_2)$, $(pair_1, pair_3)$ and $(pair_2, pair_3)$, ignoring the remaining symmetric cases. To do so, the dealer must be instructed to provide a pair to a specific player. In `CATLib`, the actions of contract automata can also feature an *address*: this is used to specify that the action must be matched by a specific service. The contract of the dealer is modified as follows: from the initial state it provides to the first player either $pair_1$ or $pair_2$. In the first case ($pair_1$ to Player 1), $pair_2$ or $pair_3$ is provided to the second player. In the second case ($pair_2$ to Player 1), $pair_3$ is provided to the second player. The composition with the pruning predicate, committed states and discarding the symmetric cases has 36 states and 37 transitions.

Summing up, we thus showed that, by using the state-space reduction techniques offered by `CATLib`, plus adequate modelling, we managed to avoid the generation of around 95% of the original state space. We also demonstrated that committed states help reduce the state space, when compared to the alternative of model checking the composition without committed states.

Orchestration heuristics In Section 4, we hypothesised that the conditional orchestration (Definition 2) can be used as an upper bound for the set of splitting orchestrations (Definition 6). Similarly, the set of refined conditional orchestrations (Definition 5) can be used as a lower bound for the set of splitting orchestrations (Definition 6), as depicted in Fig. 5.

Fig. 5 is useful for providing a graphical depiction of the relationships between various orchestration algorithms, but it lacks precision. When addressing the relations between the various orchestration syntheses, “containment” is intended in the sense of the

partial order of controllers discussed in [2]. For example, given a composed contract automaton $\mathcal{A}$ and two orchestrations $\mathcal{O}^{u}_{\mathcal{A}}$ and $\mathcal{O}^{l}_{\mathcal{A}}$, we have that $\mathcal{O}^{l}_{\mathcal{A}} \subseteq \mathcal{O}^{u}_{\mathcal{A}}$ holds whenever the "lower-bound" orchestration $\mathcal{O}^{l}_{\mathcal{A}}$ of $\mathcal{A}$ is included in the "upper-bound" orchestration $\mathcal{O}^{u}_{\mathcal{A}}$ of $\mathcal{A}$.

We first formally prove that the set of conditional orchestrations (Definition 2) includes the set of refined conditional orchestrations (Definition 5).

Theorem 1. *Let $\mathcal{A}$ be a contract automaton, let $\mathcal{O}^{con}_{\mathcal{A}}$ be its conditional orchestration (Definition 2), and let $\mathcal{O}^{ref}_{\mathcal{A}}$ be its refined conditional orchestration (Definition 5). It holds that*

$$\mathcal{O}^{ref}_{\mathcal{A}} \subseteq \mathcal{O}^{con}_{\mathcal{A}}$$

Regarding the splitting orchestration, we prove a more modest result.

Lemma 1. *There exists a contract automaton $\mathcal{A}$, with $\mathcal{O}^{con}_{\mathcal{A}}$ its conditional orchestration, $\mathcal{O}^{ref}_{\mathcal{A}}$ its refined conditional orchestration and $\mathcal{O}^{split}_{\mathcal{A}}$ its splitting orchestration (Definition 6), such that*

$$(\mathcal{O}^{con}_{\mathcal{A}} \not\subseteq \mathcal{O}^{split}_{\mathcal{A}}) \vee (\mathcal{O}^{split}_{\mathcal{A}} \not\subseteq \mathcal{O}^{ref}_{\mathcal{A}})$$

The inclusion relations between the other orchestrations (i.e., involving $\mathcal{O}^{split}_{\mathcal{A}}$) are more involved. Indeed, the conditional orchestrations are applied to the same automaton, thus enabling simpler reasoning in terms of automata containment (i.e., $\mathcal{O}^{ref}_{\mathcal{A}} \subseteq \mathcal{O}^{con}_{\mathcal{A}}$). However, the splitting orchestration is applied to an encoded automaton that contains τ transitions, whereas a (refined) conditional orchestration is applied to an automaton without τ transitions. In this case, the τ transitions in $\mathcal{O}^{split}_{\mathcal{A}}$ can be treated as silent actions, and the containment relation needs to be expressed either as trace containment or as a modal refinement. Furthermore, while $\mathcal{O}^{split}_{\mathcal{A}}$ only contains controllable and uncontrollable transitions, both $\mathcal{O}^{con}_{\mathcal{A}}$ and $\mathcal{O}^{ref}_{\mathcal{A}}$ contain controllable and semi-controllable transitions. However, for example, each semi-controllable transition in $\mathcal{O}^{con}_{\mathcal{A}}$ is evaluated to be either controllable or uncontrollable in $\mathcal{O}^{con}_{\mathcal{A}}$ (recall that the orchestration is the fixpoint of the synthesis function).

We provide an informal argument regarding the relation between the orchestrations generated through Definition 6 (splitting orchestration synthesis) and orchestrations generated through Definition 3 (conditional orchestration synthesis). We observe that conditional semi-controllability (Definition 2) necessitates the matching of a lazy necessary request everywhere else in the orchestration, even across distinct traces. In contrast, the splitting orchestration synthesis imposes a more stringent condition: the service executing the lazy necessary request must wait in a non-final state until its necessary request is eventually matched (within the same execution).

Finally, the relation between the orchestrations generated through Definition 5 (refined conditional orchestration) and the orchestrations generated through Definition 6 (splitting orchestration synthesis) can be established as follows. Employing Definition 5 (refined conditional orchestration), a service remains idle in the source state of the necessary lazy request until that specific request is matched by some other service. When multiple lazy necessary requests are enabled from the same source state (cf., e.g., Fig. 7), all such requests must eventually be matched. In contrast, under Definition 6, once the service internally determines which among the available lazy necessary requests should be matched, it is no longer required that the other lazy necessary requests must be matched. Therefore, whenever a refined conditional orchestration is non-empty, it follows that also the splitting orchestration is non-empty.

We performed a set of experiments to assess the effectiveness of utilising Definitions 2 and 5 as upper- and lower-bound heuristics for a splitting orchestration. Definition 2 necessitates a visit of the automaton at each iteration of the synthesis process to determine whether a semi-controllable transition is controllable or uncontrollable. This requirement is not present in Definition 6. On the other hand, the encoding in Definition 6 increases the size of the automata by introducing an additional state for each necessary lazy transition.

We measured the computing time of the synthesis process when employing Definitions 2, 5 and 6, using the card game. This is also useful to empirically validate the relations between $\mathcal{O}^{ref}_{\mathcal{A}}$, $\mathcal{O}^{split}_{\mathcal{A}}$ and $\mathcal{O}^{con}_{\mathcal{A}}$. However, we remark that the relations with the splitting orchestrations are currently conjectures, the formal proofs of which require further work.

The measured time is the total time required for computing the composition plus synthesising the orchestration. In this experiment, we used both committed states and the pruning predicate, but we did not employ the symmetric reduction. The composed automaton is the same in all three experiments, which only differ in the synthesis process.

The experiments were performed on a laptop Lenovo Thinkpad X1 with processor 13th Gen Intel(R) Core(TM) i7-1365U, 1800 MHz, 10 Cores, 12 Logical Processors and 32 GB of RAM. We executed the experiments within the IDE IntelliJ IDEA 2023.2.3 (Community Edition), on Windows 11 with power mode set to best performances.

The results (showed in Table 2) confirm our hypothesis. The conditional orchestration can be used as an heuristic to over-approximate the splitting orchestration. Indeed, computing the conditional orchestration is faster and only required 52 milliseconds, compared to the 241 milliseconds necessary to compute the splitting orchestration. Furthermore, although the splitting orchestration introduces additional states necessary to encode the lazy necessary transitions, the number of states of the conditional orchestration (49) is still greater than the number of states of the splitting orchestration (41). This confirms the fact that the conditional

Table 2
Comparison of different orchestrations for the card game.

	Time	*States*	*Transitions*
Conditional Orchestration	52 ms	49	54
Splitting Orchestration	241 ms	41	54
Refined Conditional Orchestration	42 ms	0	0

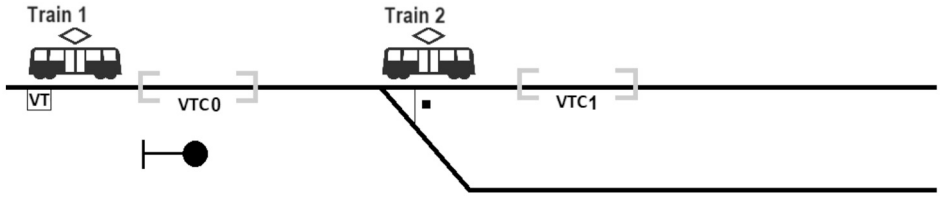


Fig. 12. The scenario taken from [15], where Train 1 is waiting to enter a junction area while Train 2 is traversing it.

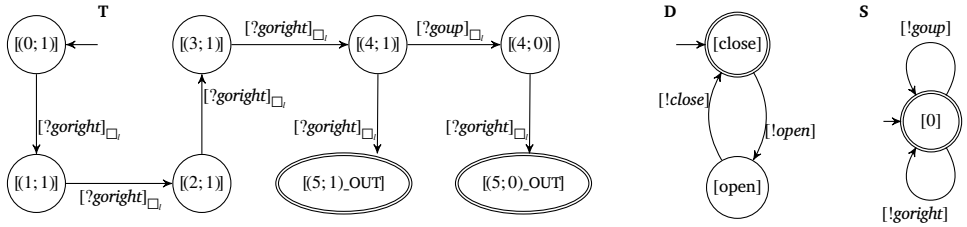


Fig. 13. The principal CA of the first **T**rain, the **D**river and the **S**emaphore control.

orchestration is an over-approximation, introducing further traces not present in the dealer's strategy (e.g., in the conditional orchestration the dealer also provides $pair_3$, and forces the players in selecting cards in the right order). It also confirms that the refined conditional orchestration is an under-approximation: in this case, the resulting orchestration is empty, and the required computing time is 42 milliseconds.

Note that we referred to the conditional orchestrations as heuristics because they approximate the optimal solution (the splitting orchestration) but can be computed in less time.

5.2.3. Railway example

We repeat this comparison on another example based on a real-world case study from the railway domain. The specific example is taken from [15,3].

In [15], an autonomous train positioning (ATP) system is analysed. In an ATP, the physical track circuits detecting the occupancy of portions of the railway track are substituted by virtual track circuits (VTCs). The VTCs are virtual positions on a map. The real position of a train is detected using a global positioning system (GPS).

In the scenario in Fig. 12, one junction area (commanded by one interlocking) is composed of two VTCs, and there is one train outside the junction area and one train inside the junction area. Train 2 is traversing its assigned route, while Train 1 is waiting at a red signal for its route to be assigned. VTC 0 is used to detect the occupation of a route, whereas VTC 1 is used to detect the release of a route. Initially, both trains are located behind the semaphore. The first train arriving (Train 2) will communicate its route to the interlocking, which will proceed to set the route. This may cause the movement of the junction point. Once the route is set, the interlocking will signal to Train 2 that the route is set by opening the semaphore. Train 2 enters the junction point and the semaphore is closed again. While Train 2 is traversing its route, the second train arriving (Train 1) will stop at the (closed) semaphore to ask for its route. The route will be assigned, the junction point moved, and the semaphore opened only after Train 2 has exited the junction area. Otherwise, the movement of the junction point could cause the derailment of Train 2 inside the junction area [15].

Fig. 13 shows three of the five principal CA used to model this scenario. The states of the **T**rain automaton represent spatial coordinates in a bidimensional map (the railway track) of where the train is moving. The train requests to move to an adjacent location through necessary lazy requests (i.e., the train is semi-controllable). This automaton models the railway track depicted in Fig. 12. The junction area is identified as the location with coordinate $x = 4$, where the train is allowed to perform the transition $[(4;1)] \xrightarrow{[?goup]_{\square_l}} [(4;0)]$ modelling the traversing of the point. Furthermore, in the experiments the location of the semaphore is set to have coordinate $x = 2$.

A **D**river automaton is used to match the requests of the train to move. Similarly, the **S**emaphore control automaton controls the semaphore by opening and closing it. We do not display the automaton of the semaphore (identical to the semaphore control in Fig. 13, but with offers switched to requests) and the automaton of the second train (identical to the automaton of the first train in Fig. 13, but with initial location $(1;1)$).

Table 3
Comparison of different orchestrations for the train example.

	Time	*States*	*Transitions*
Conditional Orchestration	40 ms	32	44
Splitting Orchestration	207 ms	90	159
Refined Conditional Orchestration	31 ms	32	44

When composing these five CA, the pruning predicate is used both to enforce strong agreement and to identify the forbidden states in the composition (if the target state of the input transition is forbidden, then the pruning predicate evaluates to true). The forbidden states are states satisfying one or more of these requirements: (i) both trains are in the same location (and are not in the final location modelling the exit from the area); (ii) one train is at the semaphore location whilst the semaphore is closed; (iii) both trains are inside the junction; (iv) a train is inside the junction area and the semaphore is opened; and (v) the semaphore is opened but no train is near it (in this case the semaphore must be closed). It is easy to see that all these requirements are invariants that can be checked on a state of the composition through the pruning predicate during the state space generation.

The orchestration of this example is a strategy for the semaphore controller to command the opening and closing of the semaphore in such a way that both (semi-controllable) trains are allowed to reach the exit while satisfying the above requirements.

Table 3 shows the comparison of the different orchestrations for the railway example. This example further corroborates our hypothesis: the two conditional orchestrations can be used as heuristics, as they are computed faster than the splitting orchestration. Indeed, while the splitting orchestration requires 207 milliseconds, the two conditional orchestrations only require 40 and 31 milliseconds. Concerning the size, the splitting orchestration has a larger number of states and transitions because each lazy necessary transition of a train is split into two transitions. We note that since the refined conditional orchestration is non-empty, also the splitting orchestration is non-empty. This is due to the refined conditional orchestration imposing stricter requirements. Therefore, in this specific example, the two conditional orchestrations are not approximations; instead, they all compute the same strategy to control the semaphore (ignoring the encoding imposed by the splitting orchestration). Note that this is not always true (cf., e.g., the card game), and that although the current implementation of the splitting orchestration synthesis has the worst performance, it is needed to solve the second research challenge and to identify the unknown set of orchestrations in Fig. 5.

RQ3 Summary: `CATLib` provides support for scalable techniques that have been demonstrated to significantly decrease the size of the generated state space. Both conditional orchestration syntheses of Definitions 3 and 5 have been proven useful as heuristics approximating the splitting orchestration synthesis of Definition 6, leading to expedited computing times.

Threats to validity For each threat, we describe the mitigations that were adopted in our experiments.

Internal validity The proposed syntheses algorithms have been implemented in `CATLib`, and experimented on various examples. All orchestrations discussed in this paper have been automatically computed by `CATLib`.

External validity We replicated the experiments on two examples, where one is a real-word case study taken from the literature [15]. The two sets of experiments produced coherent results.

Construct validity The classes `Istant`, `Duration` have been adopted to accurately measure the computing time. The implementation of the novel synthesis operators and the source code of the experiments are open source and publicly available [14].

5.3. Fourth research question: consolidating requirements

We conclude this section by addressing the fourth research question, which entails consolidating a set of requirements for effective orchestrations. Notably, since internal and external choices are now explicitly specified in contracts, the requirement of independence of choice can be removed. On the converse, the splitting orchestration synthesis implements the decoupling of choice requirement.

RQ4 Summary: The interpretation of semi-controllable transitions provided by Definition 6 discards the requirement of independence of choice, whilst confirming and implementing the decoupling of choices.

6. Conclusion

We have advanced the orchestration synthesis of contract automata by addressing a set of four research questions originally proposed in [13]. In contract automata, transitions can be either controllable or semi-controllable. We discussed the intended intuition of semi-controllable transitions, wherein services internally make the selection of these transitions, while their execution is scheduled by the orchestrator.

Initially, we refined the existing notion of orchestration synthesis in [2] that we coined *conditional* orchestration. In a conditional orchestration synthesis, a semi-controllable lazy transition can be either controllable or uncontrollable depending on given conditions. We have showed through examples how conditional orchestrations are approximations of the intended behaviour of semi-controllable transitions.

Afterwards, we proposed a new orchestration synthesis, which we called *splitting* orchestration. In this newly introduced orchestration synthesis, semi-controllable transitions are split into two concatenated transitions: one controllable and one uncontrollable. We have showed how the splitting orchestration precisely formalises the intended intuition of semi-controllability.

We have exercised these various orchestration syntheses on two different examples, a card game and a real-world case study from the railway domain. We have demonstrated through experiments that the computing time for conditional orchestrations is shorter than that for splitting orchestrations, making them practical heuristics. Various state-space reduction techniques have also been experimented with. The implementation of the new synthesis operators and the source code of the experiments are both open source and publicly available [14].

For future work, we note that the new notion of splitting orchestration necessitates updating accordingly the underlying execution support, CARE, to align it with the new contract automata specifications. This entails reducing the implementation freedom for each choice to adhere to the contract's explicit selection of the next transition. Furthermore, we conducted a series of experiments to demonstrate that the conditional orchestration can serve as heuristics to approximate the splitting orchestration, thus requiring less computing time. However, formally proving the inclusion relation with the splitting orchestration is left for future work.

CRediT authorship contribution statement

Davide Basile: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Maurice H. ter Beek:** Writing – review & editing, Visualization, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Part of this study was carried out within the MUR PRIN 2022 PNRR P2022A492B project ADVENTURE (ADVancEd iNtegraTed evalUation of Railway systEms) and the MOST – Sustainable Mobility National Research Center and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4, COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1033 17/06/2022, CN00000023. This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

Appendix A. Proofs

Theorem 1. *Let $\mathcal{A}$ be a contract automaton, let $\mathcal{O}^{con}_{\mathcal{A}}$ be its conditional orchestration (Definition 2), and let $\mathcal{O}^{ref}_{\mathcal{A}}$ be its refined conditional orchestration (Definition 5). It holds that*

$$\mathcal{O}^{ref}_{\mathcal{A}} \subseteq \mathcal{O}^{con}_{\mathcal{A}}$$

Proof. From [2, Theorem 5.4], the conditional orchestration synthesis is an instantiation of the abstract synthesis function from [2, Definition 5.1], where the pruning and forbidden predicates are instantiated as follows:

$$\begin{aligned}
\phi^{con}_{p}(t,\mathcal{O},R) &= (t \text{ is a request}) \vee (\vec{q}\,' \in R)\\
\phi^{con}_{f}(t,\mathcal{O},R) &= \nexists (\vec{q}_2 \xrightarrow{\vec{a}_2} \vec{q}_2\,') \in T^{\square}_{\mathcal{O}} : (\vec{a}_2 \text{ is a match}) \wedge (\vec{q}_2, \vec{q}_2\,' \notin \mathit{Dangling}(\mathcal{O}))\\
&\qquad \wedge (\vec{q}_{(i)} = \vec{q}_{2(i)}) \wedge (\vec{a}_{(i)} = \vec{a}_{2(i)} = a)
\end{aligned}$$

Furthermore, by [2, Proposition 6.2], to prove $\mathcal{O}^{ref}_{\mathcal{A}} \subseteq \mathcal{O}^{con}_{\mathcal{A}}$, it suffices to show that $(\phi^{con}_{p}, \phi^{con}_{f}) \leq (\phi^{ref}_{p}, \phi^{ref}_{f})$, where ϕ^{ref}_{f} and ϕ^{ref}_{p} are, respectively, the forbidden and pruning predicate of the refined conditional orchestration. By Definition 5, it holds that $\phi^{ref}_{p} = \phi^{con}_{p}$ and $\phi^{ref}_{p} = \phi^{con}_{p} \wedge \phi'$, where ϕ' expresses the condition of the second item of Definition 5. Therefore, $(\phi^{con}_{p}, \phi^{con}_{f}) \leq (\phi^{ref}_{p}, \phi^{ref}_{f})$ holds trivially since $\forall i \in \mathbb{N}\,.\,(\phi^{con}_{p}(t, \mathcal{O}^{con}_{i}, R^{con}_{i}) \Rightarrow (\phi^{ref}_{p}(t, \mathcal{O}^{ref}_{i}, R^{ref}_{i}) \vee t \notin T_{\mathcal{O}^{ref}_{i}}))$. So $\mathcal{O}^{ref}_{\mathcal{A}} \subseteq \mathcal{O}^{con}_{\mathcal{A}}$ follows by [2, Proposition 6.2]. □

Lemma 1. *There exists a contract automaton $\mathcal{A}$, with $\mathcal{O}^{con}_{\mathcal{A}}$ its conditional orchestration, $\mathcal{O}^{ref}_{\mathcal{A}}$ its refined conditional orchestration and $\mathcal{O}^{split}_{\mathcal{A}}$ its splitting orchestration (Definition 6), such that*

$$(\mathcal{O}^{con}_{\mathcal{A}} \not\subseteq \mathcal{O}^{split}_{\mathcal{A}}) \vee (\mathcal{O}^{split}_{\mathcal{A}} \not\subseteq \mathcal{O}^{ref}_{\mathcal{A}})$$

Proof. Let $\mathcal{A} = \textbf{Dealer} \otimes \textbf{Player} \otimes \textbf{Player}$ be the composed automaton from Example 7. It holds that $\mathcal{O}_{\mathcal{A}}^{split} \neq \langle\rangle$, whereas $\mathcal{O}_{\mathcal{A}}^{ref} = \langle\rangle$; therefore, $\mathcal{O}_{\mathcal{A}}^{split} \not\subseteq \mathcal{O}_{\mathcal{A}}^{ref}$.

Consider now $\mathcal{A}' = \textbf{Alice} \otimes \textbf{Bob} \otimes \textbf{Carl}$ from Example 6. It holds that $\mathcal{O}_{\mathcal{A}'}^{con} \neq \langle\rangle$, whereas $\mathcal{O}_{\mathcal{A}'}^{split} = \langle\rangle$; therefore, $\mathcal{O}_{\mathcal{A}'}^{con} \not\subseteq \mathcal{O}_{\mathcal{A}'}^{split}$. The thesis follows using either $\mathcal{A}$ or $\mathcal{A}'$. □

References

[1] A. Bouguettaya, M.P. Singh, M.N. Huhns, Q.Z. Sheng, H. Dong, Q. Yu, A.G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, Y. Wang, M.B. Blake, S. Dustdar, F. Leymann, M.P. Papazoglou, A service computing manifesto: the next 10 years, Commun. ACM 60 (4) (2017) 64–72, https://doi.org/10.1145/2983528.

[2] D. Basile, M.H. ter Beek, R. Pugliese, Synthesis of orchestrations and choreographies: bridging the gap between supervisory control and coordination of services, Log. Methods Comput. Sci. 16 (2) (2020) 9:1–9:29, https://doi.org/10.23638/LMCS-16(2:9)2020.

[3] D. Basile, M.H. ter Beek, L. Bussi, V. Ciancia, A toolchain for strategy synthesis with spatial properties, Int. J. Softw. Tools Technol. Transf. 25 (5) (2023) 641–658, https://doi.org/10.1007/S10009-023-00730-1.

[4] P.J. Ramadge, W.M. Wonham, Supervisory control of a class of discrete event processes, SIAM J. Control Optim. 25 (1) (1987) 206–230, https://doi.org/10.1137/0325013.

[5] E. Asarin, O. Maler, A. Pnueli, J. Sifakis, Controller synthesis for timed automata, IFAC Proc. 31 (18) (1998) 447–452, https://doi.org/10.1016/S1474-6670(17)42032-5.

[6] R. Ehlers, S. Lafortune, S. Tripakis, M.Y. Vardi, Supervisory control and reactive synthesis: a comparative introduction, Discrete Event Dyn. Syst. 27 (2) (2017) 209–260, https://doi.org/10.1007/s10626-015-0223-0.

[7] M. Luttenberger, P.J. Meyer, S. Sickert, Practical synthesis of reactive systems from LTL specifications via parity games, Acta Inform. 57 (1–2) (2020) 3–36, https://doi.org/10.1007/s00236-019-00349-3.

[8] P. Felli, N. Yadav, S. Sardina, Supervisory control for behavior composition, IEEE Trans. Autom. Control 62 (2) (2017) 986–991, https://doi.org/10.1109/TAC.2016.2570748.

[9] A. Camacho, M. Bienvenu, S.A. McIlraith, Towards a unified view of AI planning and reactive synthesis, in: Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'18), AAAI, 2019, pp. 58–67.

[10] D. Basile, P. Degano, G.L. Ferrari, Automata for specifying and orchestrating service contracts, Log. Methods Comput. Sci. 12 (4) (2016) 6:1–6:51, https://doi.org/10.2168/LMCS-12(4:6)2016.

[11] D. Basile, M.H. ter Beek, P. Degano, A. Legay, G.L. Ferrari, S. Gnesi, F. Di Giandomenico, Controller synthesis of service contracts with variability, Sci. Comput. Program. 187 (2020) 102344, https://doi.org/10.1016/j.scico.2019.102344.

[12] D. Basile, M.H. ter Beek, Contract automata library, Sci. Comput. Program. 221 (2022), https://doi.org/10.1016/j.scico.2022.102841, https://github.com/contractautomataproject/ContractAutomataLib.

[13] D. Basile, M.H. ter Beek, Research challenges in orchestration synthesis, in: C. Aubert, C. Di Giusto, S. Fowler, L. Safina (Eds.), Proceedings of the 16th Interaction and Concurrency Experience (ICE'23), in: EPTCS, vol. 383, 2023, pp. 73–90.

[14] D. Basile, Advancing Orchestration Synthesis for Contract Automata – Complementary Material, 2024, https://doi.org/10.5281/zenodo.11449479.

[15] D. Basile, A. Fantechi, L. Rucher, G. Mandò, Analysing an autonomous tramway positioning system with the Uppaal Statistical Model Checker, Form. Asp. Comput. 33 (6) (2021) 957–987, https://doi.org/10.1007/s00165-021-00556-1.

[16] F. Barbanera, I. Lanese, E. Tuosto, On composing communicating systems, in: C. Aubert, C. Di Giusto, L. Safina, A. Scalas (Eds.), Proceedings of the 15th Interaction and Concurrency Experience (ICE'22), in: EPTCS, vol. 365, 2022, pp. 53–68.

[17] D. Basile, P. Degano, G.L. Ferrari, E. Tuosto, From orchestration to choreography through contract automata, in: I. Lanese, A. Lluch Lafuente, A. Sokolova, H. Torres Vieira (Eds.), Proceedings of the 7th Interaction and Concurrency Experience (ICE'14), in: EPTCS, vol. 166, 2014, pp. 67–85.

[18] D. Basile, P. Degano, G.L. Ferrari, E. Tuosto, Relating two automata-based models of orchestration and choreography, J. Log. Algebraic Methods Program. 85 (3) (2016) 425–446, https://doi.org/10.1016/j.jlamp.2015.09.011.

[19] N. Yoshida, F. Zhou, F. Ferreira, Communicating finite state machines and an extensible toolchain for multiparty session types, in: E. Bampis, A. Pagourtzis (Eds.), Proceedings of the 23rd International Symposium on Fundamentals of Computation Theory (FCT'21), in: LNCS, vol. 12867, Springer, 2021, pp. 18–35.

[20] D. Kouzapas, O. Dardha, R. Perera, S.J. Gay, Typechecking protocols with Mungo and StMungo: a session type toolchain for Java, Sci. Comput. Program. 155 (2018) 52–75, https://doi.org/10.1016/j.scico.2017.10.006.

[21] R.E. Strom, S. Yemini, Typestate: a programming language concept for enhancing software reliability, IEEE Trans. Softw. Eng. 12 (1) (1986) 157–171, https://doi.org/10.1109/TSE.1986.6312929.

[22] D. Basile, M.H. ter Beek, A runtime environment for contract automata, in: M. Chechik, J.-P. Katoen, M. Leucker (Eds.), Proceedings of the 25th International Symposium on Formal Methods (FM'23), in: LNCS, vol. 14000, Springer, 2023, pp. 550–567.

[23] D. Basile, Modelling, verifying and testing the contract automata runtime environment with Uppaal, in: I. Castellani, F. Tiezzi (Eds.), Proceedings of the 26th IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION'24), in: LNCS, vol. 14676, Springer, 2024, pp. 93–110.

[24] A. Trindade, J. Mota, A. Ravara, Typestates to automata and back: a tool, in: J. Lange, A. Mavridou, L. Safina, A. Scalas (Eds.), Proceedings of the 13th Interaction and Concurrency Experience (ICE'20), in: EPTCS, vol. 324, 2020, pp. 25–42.

[25] D. Basile, CATApp, https://github.com/contractautomataproject/ContractAutomataApp.

[26] L. Bacchiani, M. Bravetti, M. Giunti, J. Mota, A. Ravara, A Java typestate checker supporting inheritance, Sci. Comput. Program. 221 (2022), https://doi.org/10.1016/j.scico.2022.102844.

[27] M.H. ter Beek, J. Carmona, R. Hennicker, J. Kleijn, Communication requirements for team automata, in: J.-M. Jacquet, M. Massink (Eds.), Proceedings of the 19th International Conference on Coordination Models and Languages (COORDINATION'17), in: LNCS, vol. 10319, Springer, 2017, pp. 256–277.

[28] M.H. ter Beek, G. Cledou, R. Hennicker, J. Proença, Can we communicate? Using dynamic logic to verify team automata, in: M. Chechik, J.-P. Katoen, M. Leucker (Eds.), Proceedings of the 25th International Symposium on Formal Methods (FM'23), in: LNCS, vol. 14000, Springer, 2023, pp. 122–141.

[29] M.H. ter Beek, R. Hennicker, J. Proença, Team Automata: overview and roadmap, in: I. Castellani, F. Tiezzi (Eds.), Proceedings of the 26th IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION'24), in: LNCS, vol. 14676, Springer, 2024, pp. 161–198.

[30] D. Basile, M.H. ter Beek, A clean and efficient implementation of choreography synthesis for behavioural contracts, in: F. Damiani, O. Dardha (Eds.), Proceedings of the 23rd International Conference on Coordination Models and Languages (COORDINATION'21), in: LNCS, vol. 12717, Springer, 2021, pp. 225–238.

[31] J. Křetínský, 30 Years of modal transition systems: survey of extensions and analysis, in: Models, Algorithms, Logics and Tools, in: LNCS, vol. 10460, Springer, 2017, pp. 36–74.

[32] D. Basile, M.H. ter Beek, A. Fantechi, S. Gnesi, Coherent modal transition systems refinement, J. Log. Algebraic Methods Program. 138 (2024) 100954, https://doi.org/10.1016/j.jlamp.2024.100954.

[33] C.G. Cassandras, S. Lafortune, Introduction to Discrete Event Systems, Springer, 2006.

[34] D. Basile, F. Di Giandomenico, S. Gnesi, P. Degano, G.L. Ferrari, Specifying variability in service contracts, in: M.H. ter Beek, N. Siegmund, I. Schaefer (Eds.), Proceedings of the 11th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'17), ACM, 2017, pp. 20–27.
[35] É. André, E. Lefaucheux, D. Lime, D. Marinho, J. Sun, Configuring timing parameters to ensure execution time opacity in timed automata, in: M.H. ter Beek, C. Dubslaff (Eds.), Proceedings of the 1st Workshop on Trends in Configurable Systems Analysis (TiCSA'23), in: EPTCS, vol. 392, 2023, pp. 1–26.
[36] M.H. ter Beek, C.A. Ellis, J. Kleijn, G. Rozenberg, Synchronizations in team automata for groupware systems, Comput. Support. Coop. Work 12 (1) (2003) 21–69, https://doi.org/10.1023/A:1022407907596.