Foundations of Collaborative DECLARE

Luca Geatti[1], Marco Montali[2], and Andrey Rivkin[3](✉)

[1] University of Udine, Udine, Italy
luca.geatti@uniud.it
[2] Free University of Bozen-Bolzano, Bolzano, Italy
montali@inf.unibz.it
[3] Technical University of Denmark, Kgs. Lyngby, Denmark
ariv@dtu.dk

Abstract. Collaborative work processes are widespread, and call for sophisticated modelling techniques to guarantee that the in-focus process is able to suitably handle all the relevant ways in which external, uncontrollable participants can influence the overall behaviour. In the presence of external actors, one needs to distinguish the internal, controllable nondeterminism of the in-focus process from the uncontrollable nondeterminism of external participants. While collaborative processes have been previously studied in the context of declarative processes, where specifications distinguish how different sources of control interact, no study along this line exists in the context of the DECLARE declarative process modeling framework. To this end, we introduce "collaborative DECLARE" (coDECLARE), where activities are assigned to the internal orchestrator or to external participants, and constraints are partitioned into conditions on how the external participants can interact with the in-focus process, and conditions that must be guaranteed by the in-focus process itself, framing the resulting specifications in style of assume-guarantee (behavioral) contracts. We discuss the conceptual and explain how central tasks such as that of DECLARE consistency and enactment have to be revised for coDECLARE. Moreover, we show how the resulting tasks can be encoded into corresponding realisability and reactive synthesis tasks for LTL specifications on finite traces.

Keywords: declarative process modelling · collaborative processes · model analysis · reactive synthesis · LTL on finite traces

1 Introduction

If we consider the very core definition of what a business or work process is, we see that *collaboration* is an essential aspect. In [22], a process is defined as:

"a collection of inter-related events, activities and decision points that involve a number of actors and objects, and that collectively lead to an outcome that is of value to at least one customer".

C. Di Francescomarino et al. (Eds.): BPM 2023 Forum, LNBIP 490, pp. 55–72, 2023.
https://doi.org/10.1007/978-3-031-41623-1_4

The definition highlights, in fact, that a process involves a number of actors, of which at least one is external to the organization. This generalizes to multiple external actors that may be involved in the process execution: from providers furnishing input material, resources, and/or services to the organization, to customers and other external stakeholders that are interested in the value produced through the process.

Collaborative work processes are hence widespread. In this work, we consider how collaboration is handled by taking the subjective perspective of one of interacting parties, singling out the process orchestrated by the in-focus party and how it relates to the behavior of other parties. Other approaches are obviously possible: for example, collaboration can be globally captured in a choreography model, from which local processes can be derived.

In the context of service interaction and execution, these aspects have been tackled when formalizing and analyzing collaborative processes [1,5,46], and in the context of realizability for multi-party choreographic processes, considering that they must be enacted in a decentralized way through interaction of the parties, which in turn call for ensuring that they locally have enough visibility of the current state of affairs when required to perform some task [20,36]. More recently, similar notions have been applied to BPMN collaborative processes, sometimes indeed reflecting the two sources of nondeterminism [13], and sometimes instead blurring them [33]. Such a blurring can be distinctively seen in widely employed Petri net encodings of BPMN [21,37], where radically different constructs such as internal decisions, event-based gateways, and boundary events are all captured as free choices in the Petri nets.

A flourishing line of research on this matter exists also for declarative processes. Several approaches have brought forward formal models for collaborative/distributed process specifications, with two main lines respectively focused on Dynamic Condition Response (DCR) Graphs [31,32] and artifact-centric formalisms [4,23]. The common theme of these works is to ensure the overall correct design and executability of distributed, collaborative processes that emerge from the composition of local declarative components. Local processes are typically composed under a cooperative assumption, where their internal non-determinism related to the choices they take is aligned with that of the others.

In our approach, we consider a different setting, where collaboration is in a way approached “cautiously”, considering that when the process of an in-focus party interacts with external participants, those external participants enact their own, at least not fully controllable independent processes. This interpretation dates back to seminal contributions on open, reactive systems [30,42], where two sources of non-determinism have to be considered. On the one hand, there is non-determinism resolved by the choices internally taken by the in-focus process; on the other hand, there is non-determinism arising from the external choices picked by the other parties. While the first is within the scope of the local orchestration, and hence can be resolved existentially, the second is uncontrollable, and thus all possibilities have to be taken into account by the process.

This delicate interplay is apparent by looking into de-facto process modelling notations, such as BPMN. There, internal control-flow structures are paired with corresponding structures dealing with external actors and events, leading to constructs such as message exchange, different types of unsolicited events, boundary events attached to exception flows, and event-driven gateways. All such structures are used to indicate routes that are taken by the process not due to internal decisions of its orchestrator, but based on which of the external events (from the set of o available ones) occurs first.

Surprisingly, collaborative processes have never been studied in the context of DECLARE, where constraints are in fact all combined together without distinguishing how they interact with the different sources of control. In fact, DECLARE [40,41] simply defines a process as a monolithic set of activities equipped with a set of constraints composed in a conjunction. No distinction is given based on which parties (or agents) control which actions. Similarly, constraints are not differentiated among those that define the context of execution (e.g., indicating under which conditions external partners can interact with the process), and those that capture the expected behavior of the in-focus process (e.g., expressing what the process should do in response to external stimuli).

In this work, we close this gap by introducing *collaborative* DECLARE (coDECLARE), where activities are assigned to the internal orchestrator or to external participants, and constraints are partitioned into conditions capturing *assumptions* on how the external participants can interact with the in-focus process, and conditions that express *guarantees* provided by the in-focus process. This reflects a peculiar characteristics when dealing with work processes or, more in general, information systems, which makes them different from general reactive systems where the environment is often assumed to be completely uncontrollable. In fact, when external stakeholders interact with a process enacted by a single party, they freely decide which task to select next among the possible ones made available to them. In this sense, the presence of external actors requires the process to define *the context of interaction*, and then to *suitably react to all possible interactions within this context.* On the one hand, the process defines the *assumptions* under which its external parties can send messages/generate events that have to be handled. On the other hand, external actors are uncontrollable, thus calling the in-focus process to be able to *guarantee* a suitable management of all possible situations within the space of possibilities defined by the context.

To handle these features, we resort to the long-standing literature on *realizability and synthesis* for temporal specifications, widely studied in AI and formal methods, and we show how to lift it for defining coDECLARE and solve key tasks related to correctness and enactment. Our main contributions are as follows:

1. We discuss why it is essential in this collaborative setting to distinguish responsibility over activities and separate constraints into assumption and guarantee constraints.
2. We define coDECLARE and discuss how central tasks such as that of DECLARE consistency and enactment have to be redefined in the light of collaborative, declarative processes.

3. We show how consistency and enactment of coDECLARE specifications can be reduced to realizability and synthesis for Linear Temporal Logic over finite traces (LTLf) , for which very mature implementations exist [6,10,27,29,48,50].

The paper is organized as follows. In Sect. 2 we review the necessary background. In Sect. 3 we introduce the framework of collaborative DECLARE. Section 4 formally defines the consistency and enactment for coDECLARE, whereas Sect. 5 shows how those can be effectively checked and computed using the existing reactive synthesis techniques. Conclusions and future directions follow.

2 A Bird-Eye-View on LTLf and DECLARE

We recall the necessary background on LTL on finite traces (LTLf), and how it is used to formalize the DECLARE language.

2.1 LTLf

LTLf is a temporal logic to express properties of finite traces.

Definition 1 (Syntax of LTLf). Given a set Σ of proposition letters, a formula ϕ of LTLf is defined as follows [18]:

$$\phi \coloneqq p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathsf{X}\phi \mid \widetilde{\mathsf{X}}\phi \mid \phi \mathsf{U} \phi$$

where $p \in \Sigma$. Formulas of LTLf over the alphabet Σ are interpreted over *finite traces* (or state sequences, or words), *i.e.*, sequences in the set $(2^{\Sigma})^{+}$. ◁

Intuitively, $\mathsf{X}\phi$ indicates *strong next*, postulating that there must exist a next state in the trace, and ϕ must hold therein. Instead, $\widetilde{\mathsf{X}}\phi$ indicates *weak next*, capturing that if a next state exists, then ϕ must hold therein. Hence, $\mathsf{X}\phi$ evaluates to false in the last state of the trace, while $\widetilde{\mathsf{X}}\phi$ evaluates to true, both regardless of ϕ. Formula $\phi_1 \mathsf{U} \phi_2$ indicates instead *until*, capturing that later in the trace (obviously, before the end of the trace) ϕ_2 must hold, and in all the states between the current one and the one where ϕ_2 holds, ϕ_1 must hold.

In the following, we will write *general finite trace semantics* to denote the interpretation under these structures. Let $\sigma = \langle \sigma_0, \ldots, \sigma_{n-1} \rangle \in (2^{\Sigma})^{+}$ be a finite trace. We define the *length* of σ as $|\sigma| = n$. With $\sigma_{[i,j]}$ (for some $0 \leq i \leq j < |\sigma|$) we denote the subinterval $\langle \sigma_i, \ldots, \sigma_j \rangle$ of σ.

Definition 2 (LTLf satisfaction relation, model, language, equivalence). Given an LTLf formula ϕ over Σ, the *satisfaction relation* of Σ by trace $\sigma \in (2^{\Sigma})^{+}$ at time $0 \leq i < |\sigma|$, denoted by $\sigma, i \models \phi$, is inductively defined as:

- $\sigma, i \models p$ iff $p \in \sigma_i$;
- $\sigma, i \models \neg p$ iff $p \notin \sigma_i$;

- $\sigma, i \models \phi_1 \vee \phi_2$ iff $\sigma, i \models \phi_1$ or $\sigma, i \models \phi_2$;
- $\sigma, i \models \phi_1 \wedge \phi_2$ iff $\sigma, i \models \phi_1$ and $\sigma, i \models \phi_2$;
- $\sigma, i \models \mathsf{X}\phi$ iff $i+1 < |\sigma|$ and $\sigma, i+1 \models \phi$;
- $\sigma, i \models \widetilde{\mathsf{X}}\phi$ iff either $i+1 = |\sigma|$ or $\sigma, i+1 \models \phi$;
- $\sigma, i \models \phi_1 \mathsf{U} \phi_2$ iff there exists $i \leq j < |\sigma|$ such that $\sigma, j \models \phi_2$, and $\sigma, k \models \phi_1$ for all k, with $i \leq k < j$.

Table 1. DECLARE patterns together with their LTLf formalization

Pattern	LTLf **formalization**
`existence`(p)	$\mathsf{F}(p)$
`coexistence`(p, q)	$\mathsf{F}(p) \leftrightarrow \mathsf{F}(q)$
`resp-existence`(p, q)	$\mathsf{F}(p) \rightarrow \mathsf{F}(q)$
`not-coexistence`(p, q)	$\neg(\mathsf{F}(p) \wedge \mathsf{F}(q))$
`absence2`(p)	$\neg\mathsf{F}(p \wedge \mathsf{XF}(p))$
`response`$(p, q_1, \ldots, q_n)$	$\mathsf{G}(p \rightarrow \mathsf{F}(\bigvee_{i=1}^{n} q_i))$
`alt-response`(p, q)	$\mathsf{G}(p \rightarrow \mathsf{X}((\neg p)\mathsf{U}q))$
`chain-response`(p, q)	$\mathsf{G}(p \rightarrow \mathsf{X}(q))$
`precedence`(p, q)	$(\neg q)W(p)$
`alt-precedence`(p, q)	$((\neg q)Wp) \wedge \mathsf{G}(q \rightarrow \widetilde{\mathsf{X}}((\neg q)Wp))$
`chain-precedence`(p, q)	$\mathsf{G}(\mathsf{X}(q) \rightarrow p)$
`succession`(p, q)	$\mathsf{G}(p \rightarrow \mathsf{F}(q)) \wedge (\neg q)W(p)$
`alt-succession`(p, q)	$\mathsf{G}(p \rightarrow \mathsf{X}((\neg p)Uq)) \wedge ((\neg q)Wp) \wedge \mathsf{G}(q \rightarrow \widetilde{\mathsf{X}}((\neg q)Wp))$
`chain-succession`(p, q)	$\mathsf{G}(p \leftrightarrow \mathsf{X}(q))$
`neg-succession`(p, q)	$\mathsf{G}(p \rightarrow \neg\mathsf{F}(q))$
`neg-chain-succession`(p, q)	$\mathsf{G}(p \rightarrow \widetilde{\mathsf{X}}(\neg q)) \wedge \mathsf{G}(q \rightarrow \widetilde{\mathsf{X}}(\neg p))$
`choice`(p, q)	$\mathsf{F}(p) \vert \mathsf{F}(q)$
`exc-choice`(p, q)	$(\mathsf{F}(p) \vert \mathsf{F}(q)) \wedge \neg(\mathsf{F}(p) \wedge \mathsf{F}(q))$

We say that σ is a *model* of ϕ (written as $\sigma \models \phi$) iff $\sigma, 0 \models \phi$. The *language* (over finite trace) of ϕ, denoted by $\mathcal{L}(\phi)$, is the set of traces $\sigma \in (2^{\Sigma})^{+}$ such that $\sigma \models \phi$. We say that two formulas $\phi, \psi \in$ LTLf are *equivalent* iff $\mathcal{L}(\phi) = \mathcal{L}(\psi)$. $\triangleleft$

As customary, we define the following abbreviations: $\mathsf{F}\phi = \top \mathsf{U} \phi$ (eventually) captures that ϕ holds at some moment in the future, $\mathsf{G}\phi = \neg\mathsf{F}\neg\phi$ (globally) captures that ϕ holds from the current state to the end of the trace, and $\phi_1 W \phi_2 = \phi_1 \mathsf{U} \phi_2 \vee \mathsf{G}\neg\phi_2$ weakens U by not necessarily requiring that ϕ_2 becomes true. Also notice that $\widetilde{\mathsf{X}}\phi = \neg\mathsf{X}\neg\phi$.

2.2 DECLARE

DECLARE is a framework [41] and a language [40] for the declarative specification of processes, enjoying flexibility by design [44]. We refer to [39] for a thorough treatment of declarative processes.

A DECLARE specification consists of a finite set of *patterns* used for constraining the allowed execution traces of the process. Each pattern is defined over a set of (atomic) actions, and has a semantics based on LTLf. Table 1 recalls the typical DECLARE patterns and their LTLf formalization.

In the context of this work, we actually support arbitrary patterns in LTLf, going beyond the patterns of Table 1. We therefore directly use LTLf formulas in place of constraint patterns.

Definition 3 (DECLARE specification). A DECLARE *specification* is a pair $\langle \mathcal{A}, \mathcal{C} \rangle$, where $\mathcal{A}$ is a finite set of *activities*, and $\mathcal{C}$ is a finite set of LTLf formulas over $\mathcal{A}$, called *constraints*. ◁

DECLARE also comes with a graphical notation. We illustrate next a DECLARE specification, which will be used throughout the entire paper.

Example 1. Consider the DECLARE specification from Fig. 1(a), shown in the standard graphical notation associated to DECLARE. The specification is a fragment of an order-to-cash process. The specification dictates that an order can be paid or canceled at most once (constraint 0..1 on `cancel` and `pay`). Whenever an order is paid, then the customer address has to be set at least once, wither before or after the payment (resp-existence(`pay`,`set`)). Upon payment either shipment or refund should eventually occur (response(`pay`,`ship` $\vee$ `refund`)). In turn, shipment and refund can only occur after payment (precedence(`pay`,`ship`) and precedence(`pay`,`refund`)). Shipment is only possible if the address has been set (precedence(`pay`,`ship`)), and the address cannot be updated anymore once shipment occurs (neg-response(`ship`,`set`)). Finally, shipment and cancelation are mutually exclusive (not-coexistence(`cancel`,`ship`)). ◁

Differently from arbitrary LTLf formulas, DECLARE assumes that every state corresponds to the execution of one and only one activity, that is, that exactly one proposition is true therein [16,25]. This leads to a semantics based on so-called *simple finite traces*.

Definition 4 (Simple finite trace). A simple finite trace over $\mathcal{A}$ is a finite trace $\sigma = \langle \sigma_0, \dots, \sigma_n \rangle \in \mathcal{A}^+$, such that for every $i \in \{0, \dots, |\sigma - 1|\}$, we have $|\sigma_i| = 1$. ◁

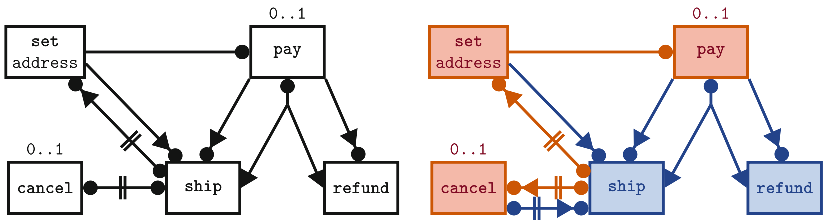
Definition 5 (DECLARE model trace). Given a DECLARE specification $\mathcal{D} = \langle \mathcal{A}, \mathcal{C} \rangle$, a simple trace σ over $\mathcal{A}$ is a *model trace* of $\mathcal{D}$ if $\sigma \models \bigwedge_{\varphi_i \in \mathcal{C}} \varphi_i$. ◁

We close by recalling that, without loss of generality, one can redefine the notion of model trace by taking as input an arbitrary LTLf trace, proviso altering the DECLARE specification with an additional special formula forcing the arbitrary trace to be simple.

Remark 1 ([24,39]). Given a DECLARE specification $\mathcal{D} = \langle \mathcal{A}, \mathcal{C} \rangle$, to check wether an arbitrary model trace σ over $\mathcal{A}$ is indeed a *model trace* of $\mathcal{D}$ we check whether $\sigma \models \psi_{simple}(\mathcal{A}) \bigwedge_{\varphi_i \in \mathcal{C}} \varphi_i$, where for a set S of propositions we define $\psi_{simple}(S) \coloneqq \mathsf{G}(\bigvee_{p \in S} p \wedge \bigwedge_{p \neq q \in S} \neg(p \wedge q))$. ◁

3 Collaborative DECLARE

We now critically assess the notion of simple traces (Definition 4), and that of DECLARE specifications (Definition 3), in the light of collaborative processes, such as actually the one introduced in Example 1.



(a) A DECLARE specification (b) A coDECLARE refinement of (a)

Fig. 1. A graphical representation of an order handling process in DECLARE (a), then refined in (b) as a coDECLARE specification. Rectangles denote actions, connectors constraints. Customer action s/constraints are in orange, those of the seller in blue. (Color figure online)

3.1 Executing a Collaborative Process

By inspecting Example 1, it is clear that the activities contained in the specification cannot be all ascribed to a single locus of execution. Instead, the process brings together two parties: a *seller* - responsible for orchestrating the order-to-cash process, and a *customer*, representing the external participant. In this light, the first important change we need to apply to the specification, is to actually assign the different activities to one of the two parties.

Example 2. In the order-to-cash process of Fig. 1, activities `set address` and `pay` are controlled by the customer, while `cancel`, `ship`, `refund` are controlled by the seller. ◁

In case of multiple external parties, for the purpose of this paper we can all model them as a single, external party, as it is typically done in the literature [34,43]. In fact, what matters is distinguishing the two different sources of nondeterminism when choosing which activity to execute: the one of the orchestrator, or that of external, uncontrollable actors. From now on, we refer to the orchestrator as *controller*, and to the external parties as *environment*.

Since the environment is uncontrollable, capturing a process execution as a simple trace (in the sense of Definition 4) is overly restrictive. Conceptually speaking, in fact, the process specification is given to controller, and therefore it is in the interest of controller to ensure that the resulting trace satisfies the specification.

In fact, controller has to *observe* what environment has done so far, so as to *suitably react*. For example, the seller may decide to behave differently depending on whether the customer has paid and later cancelled or not. This calls for moving from the notion of simple trace to that of *process strategy* for controller.

Definition 6 (Process strategy). Let $\mathcal{A}^E$ and $\mathcal{A}^C$ be two disjoint sets of activities, respectively denoting the environment and the controller activities. A *process strategy* is a function $s : (E)^+ \rightarrow P$ that for every finite sequence $\boldsymbol{e} = \langle e_0, \ldots, e_n \rangle$ of activities chosen by the environment, determines the next activity $P_n = s(\boldsymbol{e})$ executed by the controller. ◁

Notice that a process strategy respects the original DECLARE semantics of simple traces, as it enables only one activity to be executed per state.

One may wonder why the process strategy does not use the whole partial trace accumulated so far, including the activities of both the environment and the controller, but instead merely focuses on previous activities executed by environment. As we will see, more sophisticated notions of process strategies will not be needed to solve the key tasks of collaborative processes.

3.2 Satisfying the Constraints of a Collaborative Process

Now that we have a notion of process strategy, we can imagine that the orchestrator uses it to guarantee that the constraints of the DECLARE specification of interest are indeed all satisfied when concluding the execution (that is, the strategy yields, at completion time, a model trace). However, considering that the environment is uncontrollable, it turns out that this is impossible even for very trivial DECLARE specifications that assign at least one activity to environment.

Example 3. Consider the DECLARE specification of Fig. 1(a), with the activity partitioning from Example 2. The seller cannot define a process strategy that guarantees the satisfaction of all constraints, since the customer controls activities that are subject to unary constraints (0..1 on `cancel` and `pay`) or binary constraints all related to the customer (resp-existence(`pay`,`set`)). Since the customer is uncontrollable, they may issue two payments, or two cancelations, or may pay without ever setting an address. $\lhd$

Example 3 witnesses that we cannot monolithically consider all specification constraints as being under the responsibility of the controller, while leaving environment free to generate an arbitrary sequence of activities under their responsibility. In fact, as stated in the introduction, external parties need to come with some context on their freedom of choice. This is concretely reflected inside BPM systems, which expose executable activities to the external parties, and define how to handle external events, only under certain circumstances.

To reflect this requirement in our declarative setting, we give constrained freedom to the environment, by partitioning the constraints into those that must be respected by the environment to properly interact with the process, and those that the controller has to satisfy. This reflects the paradigm of *assume-guarantee contracts* [7]: under the *assumption* that the environment behaves in such a way that their constraints are satisfied, the orchestrator is bound to *guarantee* the satisfaction of their own constraints.

With this intuition in mind, we define collaborative DECLARE specifications.

Definition 7 (coDECLARE specification). A *collaborative* DECLARE (coDECLARE) specification is a tuple $\mathcal{D} = \langle \mathcal{A}^E, \mathcal{A}^C, \mathcal{C}^E, \mathcal{C}^C \rangle$, where:

- $\mathcal{A}^E$ is a finite set of *environment activities*;
- $\mathcal{A}^C$ is a finite set of *controller activities*, with $\mathcal{A}^E \cap \mathcal{A}^C = \emptyset$;

- $\mathcal{C}^E$ is a set of LTLf constraints over $\mathcal{A}^E \cup \mathcal{A}^C$ representing the *environment constraints*;
- $\mathcal{C}^C$ is a set of LTLf constraints over $\mathcal{A}^E \cup \mathcal{A}^C$ representing the *controller constraints*;

We respectively call $\mathcal{D}^a = \langle \mathcal{A}^E \cup \mathcal{A}^C, \mathcal{C}^E \rangle$ and $\mathcal{D}^g = \langle \mathcal{A}^E \cup \mathcal{A}^C, \mathcal{C}^C \rangle$ the *assumption* and *guarantee specifications* of $\mathcal{D}$. $\triangleleft$

Example 4. We refine the DECLARE specification of Fig. 1(a), considering the activity partitioning from Example 2, into the coDECLARE specification $\mathcal{D}_{order}$ of Fig. 1(b). The partitioning is done according to the following idea. To participate to the order handling process, the customer has to commit to: (i) paying and cancelling at most once, (ii) ensuring that an address is set upon payment, (iii) not updating the address nor cancelling once the seller has shipped the order. At the same time, the seller commits to: (i) shipping only if the customer sets an address and pays, (ii) refunding only if a previous customer payment exists, (iii) ensuring that shipment or refund occur whenever the customer pays, (iv) not shipping if the customer cancels the order. One can see that the vast majority of constraints present in the original DECLARE specifications have been maintained and assigned either to the environment (customer) or the controller (seller). The only exception is the not coexistence constraint relating cancelation and shipment, which is now refined into two time-oriented constraints, one assuming that the customer does not cancel after a seller's shipment (neg-response(`ship`, `cancel`)), and the other guaranteeing that the seller does not ship after a customer's cancelation ((neg-response(`cancel`, `ship`)). $\triangleleft$

We can directly lift the notion of model trace (as per Definition 5) to the case of coDECLARE. To do so, we formalize the intuition given so far: the trace must be such that whenever it satisfies the assumption on the environment, then it must satisfy the guarantee on the orchestrator. This implicitly means that executions violating the assumption on the environment are all considered model traces, as these are traces over which the environment cannot claim any guarantee. This is in line with the notion of assume-guarantee contract [7], and that of assume-guarantee synthesis [8,11,38].

Definition 8 (coDECLARE model trace). Given a coDECLARE specification $\mathcal{D} = \langle \mathcal{A}^E, \mathcal{A}^C, \mathcal{C}^E, \mathcal{C}^C \rangle$, a simple trace σ over $\mathcal{A}^E \cup \mathcal{A}^C$ is a *model trace* of $\mathcal{D}$ if, whenever it is a model trace for $\mathcal{D}^a$ (in the sense of Definition 5), then it is also a model trace for $\mathcal{D}^g$ (again in the sense of Definition 5). $\triangleleft$

Example 5. Consider the coDECLARE specification $\mathcal{D}_{order}$ of Fig. 1(b). Four model traces for $\mathcal{D}_{order}$ are $\langle$`set`, `pay`, `ship`$\rangle$, $\langle$`pay`, `set`, `refund`$\rangle$, $\langle$`pay`, `set cancel refund`$\rangle$, and $\langle$`pay set ship cancel`$\rangle$. They respectively denote a good execution where the order is shipped, an execution where the seller decides to refund for an internal problem, and an execution where the seller refunds due to a customer cancellation, and one where no refund is given in spite of cancellation since the order has been already shipped. The trace $\langle$`pay set cancel`$\rangle$ is instead not a model trace, as the seller should refund. $\triangleleft$

The main open question now is: how can the orchestrator ensure that an ongoing execution for a coDECLARE model eventually leads to a proper, model trace? The challenge here is that, even if the environment is constrained by their assumption specification, it still has a (constrained) freedom to decide which environment activities are executed, and in which order. Hence, to be able to properly execute the specification, the orchestrator must have a strategy (in the sense of Definition 6) to guarantee that no matter how the environment behaves within the space given by the assumption specification, then the execution is progressed and finally stopped by satisfying the guarantee specification. This is tackled in the next section.

4 Consistency and Enactment of coDECLARE

To tackle the problem of enactment of coDECLARE specifications, we start pointing out the striking similarity with the long-standing problem of *synthesis from declarative specification*, which dates back to Church [12]. In summary, given a declarative specification, one can define two distinct problems. The first concerns *verification*, and is about checking the correctness of the specification, namely whether the specification has a satisfying assignment (which, in the case of linear temporal specifications, means a trace). A different problem is that of *synthesis*, which deals with deriving a correct-by-construction program (in the shape, *e.g.*, of a Mealy or Moore machine, I/O-transducer, or circuit) that realizes the specification and makes it possible to execute it. Extensive research has been conducted on different synthesis settings, considering in particular closed and *open* (also called *reactive*) systems, starting from the seminal contributions by Harel, Pnueli and Rosner [30,42]. In the reactive setting, using the same terminology adopted here, the system (referred to as controller) interacts with an uncontrollable environment, which, in turn, can affect the behavior of the controller. Reactive synthesis is hence modeled as a two-player game between Controller, whose aim is to satisfy the formula, and Environment, who tries to violate it. The objective of the synthesis task is then to synthesize a program for Controller indicating which actions the Controller should take to guarantee the satisfaction of the declarative specification of interest, no matter what are the actions taken by Environment. This problem was originally studied in [12] and solved in [9], and for LTL specifications in particular it was shown to be 2EXPTIME-complete [43,45]. The high theoretical complexity and practical infeasibility of the original approach, led to a plethora of studies focused on settings more amenable to effective synthesis algorithms, one of the most important being synthesis for LTLf- thus considering *finite traces* [28].

In this section, we connect such long-standing literature with coDECLARE, in particular defining consistency and (automatic) enactment for coDECLARE by adapting to our setting the well-established notions of realizability and synthesis from declarative specifications.

4.1 Realizability over Simple Traces

We start by considering the *realizability* task [43] for LTLf formulas, in the setting where executions correspond to simple traces (cf. Definition 4). Intuitively, given an LTLf formula ϕ over two sets of controllable $\mathcal{C}$ and uncontrollable $\mathcal{U}$ variables (s.t. $\mathcal{C} \cap \mathcal{U} = \emptyset$), we have that ϕ is realizable if there exists a strategy for controller that, no matter the choices made by the environment regarding the variables in $\mathcal{U}$ to set true, chooses truth assignments to variables in $\mathcal{C}$ so that ϕ is satisfied. Hereinafter, we talk about variables in the context of general definitions, and activities in the context of coDECLARE and processes, and use $\mathcal{A}^E$ and $\mathcal{A}^C$ as uncontrollable and controllable variables, respectively.

To adapt realizability to our setting, we use process strategies from Definition 6. Since realizability is usually tested using a two-player game between the controller and environment, we postulate that such strategies are applied in the strictly alternating way, and that the environment always starts first. These assumptions will be clarified later.

Definition 9 (Realizability over simple traces). Let ϕ be an LTLf formula over $\mathcal{A}$. ϕ is *realizable over simple finite traces* iff there exists a process strategy $s : (\mathcal{U})^+ \to \mathcal{C}$ such that, for any infinite sequence $\mathcal{U} = \langle \mathcal{U}_0, \mathcal{U}_1, \ldots \rangle \in (\mathcal{U})^\omega$ of actions chosen by the environment, there exists $k \in \mathbb{N}$ such that $\mathtt{simres}(s, \mathcal{U})_{[0,k]} \models \phi$.[1]
$\triangleleft$

First and foremost, notice that the way the process strategy starts and completes is perfectly compatible with the notion of a business process. On the one hand, every process instance starts because of an activity triggered by the environment. On the other hand, the power to decide when an execution should be stopped is of the controller: it is in fact the internal orchestration mechanism that defines when a process instance reaches a final state. We now comment on strict alternation. First, the fact that every step comes with just a single chosen activity is in line with the notion of simple trace. Second, imposing alternation does not incur in any loss of generality, as we can equip both actors with a *no-op activity* whose purpose is simply to relinquish control back to the other actor. In our running example (cf. 1(b)), this is for example useful for controller to wait that the customer sets their address when the customer indeed triggers a payment without a prior execution of `set`.

4.2 Consistency and Orchestration

To ensure that a coDECLARE specification is consistent, we need to ensure that controller can define a strategy that yields a model trace – as per Definition 8. Considering Definition 10, we thus get the following.

[1] Here, $\mathtt{simres}(s, \mathcal{U}) = \langle \mathcal{U}_0, s(\langle \mathcal{U}_0 \rangle), \mathcal{U}_1, s(\langle \mathcal{U}_0, \mathcal{U}_1 \rangle), \ldots \rangle$ is the state sequence resulting from the strict alternation between the choices made by the environment and those made by the strategy s.

Definition 10 (Consistency).
A coDECLARE specification $\mathcal{D} = \langle \mathcal{A}^E, \mathcal{A}^C, \mathcal{C}^E, \mathcal{C}^C \rangle$ is *consistent* if the LTLf formula $\bigwedge_{\varphi_i \in \mathcal{C}^E} \varphi_i \rightarrow \bigwedge_{\psi_j \in \mathcal{C}^C} \psi_j$ is realizable over simple finite traces. ◁

A process strategy witnessing consistency can be effectively seen as an *orchestration mechanism* for controller: it defines a specific behaviour for controller ensuring that, whenever environment behaves in accordance to the assumption specification, the resulting reactions yield a simple trace satisfying the guarantee specification. Obviously, for a consistent specification, many different process strategyes may exist, resulting in different orchestration mechanisms for the controller. We show this in our running example.

Example 6. Consider the coDECLARE specification $\mathcal{D}_{order}$ from Fig. 1(b). Multiple process strategies exist for the seller. We show two. The first is an *always refund* strategy:

- The seller simply generates a no-op, unless the customer pays.
- As soon as the customer pays, the seller immediately reacts by refunding.

The second is a *ship as soon as possible* strategy:

- The seller simply generates a no-op, unless the customer pays.
- If the customer pays, the seller checks whether the customer has already set an address. If so, then the seller immediately ships. If not, then the seller waits for further activities executed by the seller. In particular, since the customer operates under the assumption that an address must eventually be set:
  - if the customer sets the address, the seller immediately ships afterwards;
  - if the customer cancels and only later sets the address, the seller reacts to the cancelation by refunding.

Obviously, many other process strategies exist. For example, *ship as soon as possible* may be turned into a more cautious strategy where, instead of immediately shipping whenever there are the conditions for doing so, seller waits for a while to see whether customer intends to cancel. ◁

Example 6 may show that some of the process strategies for the controller (such as the *always refund* strategy) are unintended. This should not be seen as a technical limitation of our approach, but rather as the same issue of typical *under-specification* problems arising in standard DECLARE, a well-known problem that actually pervades declarative modelling languages in general. In fact, additional constraints could be added to cut off some unintended process strategies for the controller, as discussed next.

Example 7. Consider Example 6. We may want to ensure that the seller cannot use *always refund* as an orchestration mechanism. A possible way to do so would be to constrain that the seller only refunds upon an explicit cancelation of the customer, in particular when this is triggered after a payment. This could be done by adding to the guarantee specification a further constraint resp-existence(`refund`, `cancel`). Interestingly, in every possible execution strategy for seller, this constraint will be interpreted as the more restrictive constraint precedence(`cancel`, `refund`); in fact, since `cancel` is under the control of the customer, the seller can only guarantee to satisfy resp-existence(`refund`, `cancel`) by first waiting that the customer indeed cancels, as refunding before a cancellation may lead to a violation of the constraint (if the customer decides not to cancel, which they can legitimately do). $\triangleleft$

5 Encoding into LTLf Realizability

In this section, we show how coDECLARE consistency can be checked using the standard decision procedure from the literature on LTLf realizability [19], also using the same technique to extract actual process strategies for orchestration.

To do so, we have to resolve a mismatch between the definition of realizability over simple traces (in the sense of Definition 10), and the general notion of LTLf realizability. The mismatches are that in LTLf realizability: a there is no simple trace semantics, and thus strategies are deciding on sequences of sets of actions; b strict alternation is not required and there is no need to secure exclusive control of only one player over a state in the game runs.

Definition 11 (LTLf strategy, realizability [19]**).** Given sets $\mathcal{C}$ and $\mathcal{U}$ as in the previous section, a *strategy* is a function $s : (2^{\mathcal{U}})^{+} \to 2^{\mathcal{C}}$. An LTLf formula ϕ over $\mathcal{U} \cup \mathcal{C}$ is *realizable* if there is a strategy s s.t. for every infinite sequence $\mathcal{U} = \langle \mathcal{U}_0, \mathcal{U}_1, \ldots \rangle \in (2^{\mathcal{U}})^{\omega}$, there exists $k \in \mathbb{N}$ s.t. $\texttt{res}(s, \mathcal{U})_{[0,k]} \models \phi$, where $\texttt{res}(s, \mathcal{U}) = \langle \mathcal{U}_0 \cup s(\langle \mathcal{U}_0 \rangle), \mathcal{U}_1 \cup s(\langle \mathcal{U}_0, \mathcal{U}_1 \rangle), \ldots \rangle$ is the trace resulting from reacting to $\mathcal{U}$ according to s. $\triangleleft$

Given an LTLf formula ϕ, the decision procedure for checking realizability of ϕ performs the following steps [19]. First, build a non-deterministic finite automaton § for ϕ and determinise it. Then, play a reachability game using § as the arena so as to check whether the process can reach a final state of the automaton (see [34] for more details on the actual procedure for this step). If this is the case, then ϕ is realizable, otherwise it is not.

Notice that the first step, in the worst case, takes double exponential time: § has at most exponentially many states [19] and the standard subset construction algorithm for determinization of § will require potentially lead to exponential blow-up in the number of states of the resulting deterministic automaton. The reachability game can be solved in polynomial time in the size of automaton [14] and thus does not affect the overall complexity. Notice also that if ϕ is shown to be realizable, then a strategy s can be extracted [34]. In practice, using various heuristics, the first step can be computed efficiently for DECLARE [15,47].

An important step towards reducing the consistency check (and in turn synthesis of process strategies for orchestration) to realizability (and synthesis) of LTLf over finite traces is in enforcing the assumptions made in Sect. 4 on the two-player game used for realizability checking. To this end, we define the following auxiliary LTLf formulas, making sure that (i) the game of the play is a simple trace; (ii) the environment plays at all even states; (iii) the controller plays at all odd states. The formulas are as follows:

$$\psi_{env}(\mathcal{U}) \coloneqq \bigvee_{u \in \mathcal{U}} u \wedge \mathsf{G}(\bigvee_{u \in \mathcal{U}} u \rightarrow (\bigwedge_{u \neq u' \in \mathcal{U}} \neg(u \wedge u') \wedge \widetilde{\mathsf{X}}(\bigwedge_{u \in \mathcal{U}} \neg u \wedge \widetilde{\mathsf{X}} \bigvee_{u \in \mathcal{U}} u)))$$

$$\psi_{con}(\mathcal{C}) \coloneqq \bigwedge_{c \in \mathcal{C}} \neg c \wedge \mathsf{G}(\bigwedge_{c \in \mathcal{C}} \neg c \rightarrow \widetilde{\mathsf{X}}(\bigvee_{c \in \mathcal{C}} c \wedge \bigwedge_{c \neq c' \in \mathcal{C}} \neg(c \wedge c') \wedge \widetilde{\mathsf{X}} \bigwedge_{c \in \mathcal{C}} \neg c)))$$

We use here the weak next (i.e., $\widetilde{\mathsf{X}}$) operator to ensure that the two-player game used for checking the realizability of these formulas can stop at any iteration.

The following theorem from [26] shows how we can recast the realizability technique discussed above to the case if coDECLARE specifications.

Theorem 1 ([26]). *Let $\mathcal{D} = \langle \mathcal{A}^E, \mathcal{A}^C, \mathcal{C}^E, \mathcal{C}^C \rangle$ be a* coDECLARE *specification. It holds that $\mathcal{D}$ is consistent iff the* LTLf *formula $\psi_{simple}(\mathcal{A}^E \cup \mathcal{A}^C) \wedge \psi_{con}(\mathcal{A}^C) \wedge ((\psi_{env}(\mathcal{A}^E) \wedge \mathcal{C}^E) \rightarrow \mathcal{C}^C)$ is realizable.* $\triangleleft$

Tool Support. The formula from Theorem 1 can be given as input to the realizability algorithm discussed above. This makes it possible to use any off-the-shelf tool for LTLf synthesis for the purpose of coDECLARE consistency and orchestration. This paves the way towards a direct implementation of our approach. In fact, since the introduction of the LTLf reactive synthesis problem [19], several optimized tools have been developed for solving this problem. Among all, we mention the `Syft` [51] and the `Cynthia` tools [17]. LTLf reactive synthesis is an active area of research: this is witnessed also by the organization of an annual competition (SYNTCOMP [34,35]). We thus expect that the practical efficiency of LTLf synthesis tools reflect also to coDECLARE enactment.

6 Conclusions

We have introduced a novel framework to capture *collaborative, declarative processes* specified in DECLARE, where the process orchestrator (i.e., the controller) must be able to suitably handle the interaction with uncontrollable external parties (i.e., the environment). We have described how this framework, called coDECLARE, can be naturally framed in an *assume-guarantee* style, where the following behavioral contract is stipulated by the controller and environment: under the assumption that the environment behaves according to an *assumption* DECLARE specification, the controller ensures to react by satisfying a *guarantee* DECLARE specification. We have shown, both foundationally and in terms of algorithmic support, how this framework can be connected to the well-studied framework of realizability and synthesis for LTLf specifications.

The natural, next step is to leverage this connection and provide a proof-of-concept implementation for consistency checking and process strategy generation for orchestration in coDECLARE, calling different back-end LTLf realizability/synthesis tools. We are also studying that when the LTLf formulas of interest have the shape of DECLARE patterns, better complexity bounds for solving these problems can be obtained [26]. We also foresee three interesting foundational lines of research starting from the basis provided here. The first concerns the definition of variants of consistency/orchestration for coDECLARE, in the case where the overall specification turns out to be unrealizeable. To this end, so-called best-effort strategies have been introduced [2]. However, while in our setting assume-guarantee specifications can be treated by constructing an implication formula, this is not true anymore in the case of best-effort strategies, and more sophisticated notions of synthesis under assumption have to be studied [3]. The second line concerns the actual process strategies generated from a coDECLARE specification. As discussed in the paper, each strategy defines *a particular way* for controller to ensure that whenever the environment behaves according to the assumption specification, then the guarantee specification is satisfied. In a BPM context, it would be definitely of interest to lift synthesis to a more general orchestration mechanism, where all possible strategies are combined, allowing the controller to decide at runtime, step-by-step, which specific strategy to follow. This is akin to maximally permissive strategies [49], but novel research is needed to represent them by natively dealing with concurrency and other typical control-flow patterns. A last line is to consider different notions of collaboration when dealing with DECLARE specifications. A significant setting, which departs from the one tackled here, is that where collaboration is approached in a choreographic way, observing interaction from an external point of view, and considering all the interacting parties are equally standing. This was partly studied in some seminal works [39, Chapter 8], [40], but not further developed so far.

Acknowledgements. This research has been partially supported by the UNIBZ project ADAPTERS and PRIN MIUR project PINPOINT Prot. 2020FNEB27.

References

1. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty contracts: agreeing and implementing interorganizational processes. Comput. J. **53**(1), 90–106 (2010)
2. Aminof, B., De Giacomo, G., Rubin, S.: Best-effort synthesis: doing your best is not harder than giving up. In: Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI), pp. 1766–1772. ijcai.org (2021)
3. Aminof, B., Giacomo, G.D., Murano, A., Rubin, S.: Planning under LTL environment specifications. In: Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS), pp. 31–39. AAAI Press (2019)
4. Badouel, É., Hélouët, L., Kouamou, G.E., Morvan, C.: A grammatical approach to data-centric case management in a distributed collaborative environment. In: Wainwright, R.L., Corchado, J.M., Bechini, A., Hong, J. (eds.) Proceedings of SAC 2015, pp. 1834–1839. ACM (2015)
5. Baldoni, M., Baroglio, C., Martelli, A., Patti, V.: A priori conformance verification for guaranteeing interoperability in open environments. In: Dan, A., Lamersdorf,

W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 339–351. Springer, Heidelberg (2006). https://doi.org/10.1007/11948148_28
6. Bansal, S., Li, Y., Tabajara, L.M., Vardi, M.Y.: Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 9766–9774. AAAI Press (2020)
7. Benveniste, A., et al.: Contracts for System Design. Now Foundations and Trends (2018)
8. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive (1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012)
9. Buchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. In: Mac Lane, S., Siefkes, D. (eds) The Collected Works of J. Richard Büchi. Springer, New York (1990). https://doi.org/10.1007/978-1-4613-8928-6_29
10. Camacho, A., Baier, J.A., Muise, C.J., McIlraith, S.A.: Finite LTL synthesis as planning. In: Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018), pp. 29–38. AAAI Press (2018)
11. Chatterjee, K., Henzinger, T.A.: Assume-guarantee synthesis. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 261–275. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_21
12. Church, A.: Logic, arithmetic, and automata. In: Proceedings of International Congress of Mathematicians. vol. 1962, pp. 23–35 (1962)
13. Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F., Vandin, A.: A formal approach for the analysis of BPMN collaboration models. J. Syst. Softw. **180**, 111007 (2021)
14. De Alfaro, L., Henzinger, T.A., Kupferman, O.: Concurrent reachability games. Theor. Comput. Sci. **386**(3), 188–217 (2007)
15. De Giacomo, G., De Masellis, R., Maggi, F.M., Montali, M.: Monitoring constraints and metaconstraints with temporal logics on finite traces. ACM Trans. Soft. Eng. and Method. **31**(4), 1–44 (2022)
16. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: insensitivity to infiniteness. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 1027–1033. AAAI Press (2014)
17. De Giacomo, G., Favorito, M., Li, J., Vardi, M.Y., Xiao, S., Zhu, S.: LTLF synthesis as and-or graph search: Knowledge compilation at work. In: Thirty-First International Joint Conference on Artificial Intelligence (IJCAI) (2022)
18. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. Association for Computing Machinery IJCAI 2013, pp. 854–860. IJCAI/AAAI (2013)
19. De Giacomo, G., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (AAAI), pp. 1558–1564. AAAI Press (2015)
20. Decker, G., Weske, M.: Local enforceability in interaction petri nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 305–319. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75183-0_22
21. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008)
22. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, Second Edition. Springer (2018). https://doi.org/10.1007/978-3-662-56509-4

23. Eshuis, R., Hull, R., Sun, Y., Vaculín, R.: Splitting GSM schemas: a framework for outsourcing of declarative artifact systems. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 259–274. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40176-3_22
24. Fionda, V., Greco, G.: LTL on finite and process traces: complexity results and a practical reasoner. J. Artif. Intell. Res. **63**, 557–623 (2018)
25. Fionda, V., Guzzo, A.: Control-flow modeling with declare: behavioral properties, computational complexity, and tools. IEEE Trans. Knowl. Data Eng. **32**(5), 898–911 (2019)
26. Geatti, L., Montali, M., Rivkin, A.: Reactive synthesis for DECLARE via symbolic automata. CoRR abs/2212.10875 (2022)
27. Giacomo, G.D., Favorito, M., Li, J., Vardi, M.Y., Xiao, S., Zhu, S.: LTLF synthesis as AND-OR graph search: Knowledge compilation at work. In: Thirty-First International Joint Conference on Artificial Intelligence (IJCAI 2022), pp. 2591–2598. ijcai.org (2022)
28. Giacomo, G.D., Masellis, R.D., Montali, M.: Reasoning on LTL on finite traces: Insensitivity to infiniteness. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, pp. 1027–1033. AAAI Press (2014)
29. Giacomo, G.D., Stasio, A.D., Vardi, M.Y., Zhu, S.: Two-stage technique for LTLf synthesis under LTL assumptions. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR 2020), pp. 304–314 (2020)
30. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (eds) Logics and Models of Concurrent Systems. NATO ASI Series, vol 13, pp. 477–498. Springer, Berlin, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82453-1_17
31. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T.: Designing a cross-organizational case management system using dynamic condition response graphs. In: 2011 IEEE 15th International Enterprise Distributed Object Computing Conference (EDOC 2011), pp. 161–170. IEEE Computer Society (2011)
32. Hildebrandt, T.T., Slaats, T., López, H.A., Debois, S., Carbone, M.: Declarative choreographies and liveness. In: Pérez, J.A., Yoshida, N. (eds.) FORTE 2019. LNCS, vol. 11535, pp. 129–147. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21759-4_8
33. Houhou, S., Baarir, S., Poizat, P., Quéinnec, P.: A first-order logic semantics for communication-parametric BPMN collaborations. In: Hildebrandt, T., van Dongen, B.F., Röglinger, M., Mendling, J. (eds.) BPM 2019. LNCS, vol. 11675, pp. 52–68. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26619-6_6
34. Jacobs, S., et al.: The first reactive synthesis competition. Int. J. Soft. Tools for Tech. Transf. **19**(3), 367–390 (2017)
35. Jacobs, S., et al.: The reactive synthesis competition (syntcomp): 2018–2021. arXiv preprint arXiv:2206.00251 (2022)
36. Lohmann, N., Wolf, K.: Realizability is controllability. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 110–127. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14458-5_7
37. Lomidze, G., Schuster, D., Li, CY., van Zelst, S.J.: Enhanced transformation of bpmn models with cancellation features. In: Almeida, J.P.A., Karastoyanova, D., Guizzardi, G., Montali, M., Maggi, F.M., Fonseca, C.M. (eds) Enterprise Design, Operations, and Computing. EDOC 2022. Lecture Notes in Computer Science. vol 13585, pp. 128–144. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17604-3_8

38. Maoz, S., Sa'ar, Y.: Assume-guarantee scenarios: semantics and synthesis. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 335–351. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_22
39. Montali, M.: Specification and Verification of Declarative Open Interaction Models. LNBIP, vol. 56. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14538-4
40. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographies. ACM Trans. Web **4**(1), 1–62 (2010)
41. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC), pp. 287–300. IEEE Computer Society (2007)
42. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 179–190. ACM Press (1989)
43. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ausiello, G., Dezani-Ciancaglini, M., Della Rocca, S.R. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989). https://doi.org/10.1007/BFb0035790
44. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods. Springer, Technologies (2012)
45. Rosner, R.: Modular synthesis of reactive systems. Ph.D. thesis, PhD thesis, Weizmann Institute of Science (1992)
46. Stahl, C., Wolf, K.: Deciding service composition and substitutability using extended operating guidelines. Data Knowl. Eng. **68**(9), 819–833 (2009)
47. Westergaard, M.: Better algorithms for analyzing and enacting declarative workflow languages using LTL. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 83–98. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23059-2_10
48. Xiao, S., Li, J., Zhu, S., Shi, Y., Pu, G., Vardi, M.Y.: On-the-fly synthesis for LTL over finite traces. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 6530–6537. AAAI Press (2021)
49. Zhu, S., De Giacomo, G.: Synthesis of maximally permissive strategies for ltlf specifications. In: Thirty-First International Joint Conference on Artificial Intelligence (IJCAI), pp. 2783–2789. ijcai.org (2022)
50. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: A symbolic approach to safety LTL synthesis. In: HVC 2017. LNCS, vol. 10629, pp. 147–162. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70389-3_10
51. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic LTLf synthesis. arXiv preprint arXiv:1705.08426 (2017)