DACEO: Declarative Asynchronous Choreographies with General Data-Dependent Event-Ordering and Objects

Tilman Zuckmantel(✉), Yongluan Zhou, Boris Düdder, and Thomas Hildebrandt

University of Copenhagen, Copenhagen, Denmark
tizu@di.ku.dk

Abstract. We provide a formal modeling language, DACEO, for declarative asynchronous choreographies with general data-dependent message ordering and data objects. The language is equipped with execution semantics, and thereby, it can be used to specify the semantic and support monitoring of asynchronous distributed event-based systems in which a total ordering of message delivery cannot be assumed. DACEO models are graphs, which extends previous work on synchronous declarative choreographies based on Dynamic Condition Response Graphs and their operational semantics by adding asynchronous message channels with general data-dependent ordering constraints as well as general objects and data activities. The addition of objects builds on recent work on object-centric Dynamic Condition Response Graphs. We show that the DACEO Graphs can be encoded as the more basic Synchronous Object-centric Dynamic Condition Response Choreographies, preserving the semantics. We motivate DACEO by demonstrating its applicability in describing data-dependent ordering constraints between messages using a running example that shows how a system with causal consistency requirements can be specified. The marketplace is built on a benchmark for microservices that relies on asynchronous, event-based communication.

Keywords: Formal Modeling Language · Message Ordering · Declarative Asynchronous Choreographies · Event-Based Systems

1 Introduction

With the rise of modern architecture styles like microservice architecture, distributed applications increasingly follow the database-per-service pattern. In this model, each microservice maintains its own independent data store, improving scalability and modularity. Additionally, microservices rely on asynchronous

Published by springer Nature Switzerland AG 2025
C. Di Giusto and A. Ravara (Eds.): COORDINATION 2025, LNCS 15731, pp. 197–216, 2025.
https://doi.org/10.1007/978-3-031-95589-1_10

communication to propagate messages between components. This communication pattern enables services to operate independently without blocking, increasing system responsiveness. However, the decentralized data management and asynchronous communication introduce new consistency challenges, especially when transactions span multiple microservices, leading to potential cross-service data inconsistencies [10,15,16]. Unlike traditional monolithic systems, where database systems are responsible for ensuring data consistency via mechanisms like ACID transactions and consistent data replications, microservice architectures shift the responsibility of ensuring data consistency to the application developer [10,16], requiring developers to be aware of cross-service dependencies that may impact correctness.

A consistency model that can be efficiently enforced in distributed systems, where events between microservices are processed without ordering guarantees, is causal consistency [2,17]. It provides a good trade-off between scalability and consistency level. Causal consistency means that operations respect causal relationships, even in the presence of concurrent updates. Unlike strong consistency models, which enforce a total order on all operations, causal consistency ensures that if one event causally depends on another, then all processes must observe them in that order. However, events that are independent can be seen in different orders by different nodes. It provides stronger consistency guarantees than eventual consistency while still maintaining scalability.

While being a desired consistency model in microservices, it is non-trivial to reason about causal consistency, mainly because of the asynchronous event exchange [10,15,16]. To illustrate this challenge, Fig. 1 shows an example of a causal inconsistency scenario in an online marketplace adopted from [15].

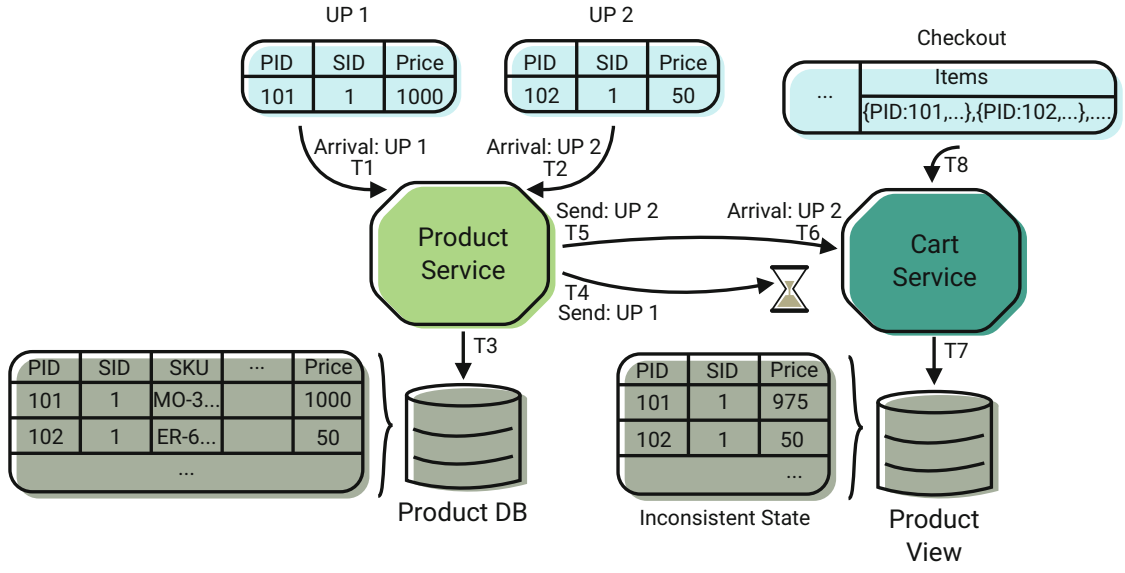


Fig. 1. Motivating example of a casual inconsistency scenario in a microservice oriented market place.

In this scenario, we show two microservices within an online marketplace: the *Product Service*, which maintains the primary dataset of products, and the *Cart Service*, which tracks customer carts. To optimize performance, the *Cart Service* also maintains a partially replicated view of the product dataset, containing only the fields relevant for price updates, such as product id (*PID*), seller id (*SID*), and *Price*. This allows the *Cart Service* to perform fast look-ups of pricing information during a checkout request without querying the full product database. Our causal consistency requirements are that orders between price updates targeting the same seller must be preserved between *Product Service* and *Cart Service*. In this scenario, two price updates (*UP 1* and *UP 2*) for different products are requested by the same seller at global times $T1$ and $T2$, respectively, where $T1 \prec T2$, meaning *UP 1* must precede *UP 2*. For example, the seller wants to increase the price of the product by 101 while decreasing the price of the product by 102 to maximize the expected profit. After updating the *Product Service* database ($T3$), the *Product Service* generates and sends a new price update event containing the same payload as the initial request, first *UP 1* ($T4$), followed by *UP 2* ($T5$) to the *Cart Service*. However, the updates arrive out of order, and only *UP 2* reaches the *Cart Service* ($T6$) and is applied ($T7$), while *UP 1* is delayed. This violation of the expected order of price updates results in an inconsistent view of the product data within the *Cart Service*. When a checkout request is issued at time $T8$, it processes the transaction based on inconsistent price information, leading to incorrect pricing and potential financial losses for sellers.

This example illustrates that enforcing a strict FIFO order on all price updates is unnecessarily restrictive, but allowing completely unordered message processing leads to inconsistencies. Instead, what is needed is a way to precisely specify ordering constraints that respect the required causal relationships, such as ensuring that price updates are applied per seller without enforcing unnecessary global constraints. While various implementation strategies exist to handle event ordering in practice, there is no dedicated formalism that allows specifying such constraints in an abstract, declarative way. To explore how existing modeling approaches handle related challenges, we turn to formal models of distributed processes and messaging protocols.

Related Work: In the area of process models, there is currently a trend towards modeling processes based on activity decisions and decisions driven by data, leading to so-called data-aware processes [6]. As a result, extensions to BPMN have been developed that integrate data and enable SQL-like queries [4,12]. Similarly, declarative modeling approaches such as DCR Choreograhies [13] and Declare [18] have been extended to support data-dependent constraints, together with a spawn relation which enables the spawning of data objects through activities [7,8]. However, these extensions still have serious limitations, such as the fact that it is not possible to have relationships between specific spawned objects, but only between all objects, and that values carried by objects cannot be assigned during creation. DCR graphs are supporting The DCR Choreographies and DCR

Graphs with data in general are supported by commercial design and execution tools[1] that are widely used in the public sector in Denmark. Also, a recent strand of work on object-centricity [1,5,9,11] for process models is providing means for specifying more complex data dependencies in process models. While these models and tools establish a foundation for integrating data-driven decisions into processes, they primarily focus on synchronously orchestrated workflows and do not explicitly address the challenges of asynchronous communication. Moreover, On the other hand, formal models such as multiparty asynchronous session types [14] and choreography automata [3] ensure that distributed components adhere to a specified messaging protocol, verifying properties such as deadlock freedom, and liveness. These approaches assume that communication occurs over FIFO-ordered message queues, which means that messages are received in the order they were sent. While this assumption simplifies reasoning about protocol correctness, real-world distributed systems often exhibit more complex message-ordering patterns. In addition, FIFO ordering guarantees are not necessary in many cases and are, therefore, too strict. In our example in Fig. 1, it is too strict to guarantee FIFO semantics for all price update events. Here, it is sufficient to order the events according to their seller id, so that for each seller, it is guaranteed that all price updates are seen. In fact, the exact demands on event ordering depend on the policies of a system configuration and their causal relationships.

Contributions: Therefore, to the best of our knowledge, no existing process model declaratively specifies event ordering constraints based on data in asynchronous channels between distributed components. To fill the gap, we introduce in this paper an extension of the declarative process model of Dynamic Condition Response (DCR) Graphs with data and chorographies based on this model, originally introduced in [13]. Intuitively, choreographies based on DCR graphs express interactions between concurrently running processes in a declarative way by stating relationships between the activities within the processes. In addition, data values can be assigned and propagated between activities to allow data-dependent decisions during the execution. Although being an expressive model for defining data-dependent processes in a declarative way, the model assumes synchronous communication between participants and lacks a mechanism to express asynchronous communication. Beyond this limitation, database entries in our example are being added and updated dynamically. However, DCR choreographies, while being able to express data in activities, cannot model the dynamic addition of data and life cycles of data objects. To overcome these challenges, we make the following contributions:

1. We extend DCR graphs by introducing a notion of objects, hereby following the popular approach of object-centricity [1,5,9,11] for process models. Objects can be dynamically added during the execution of a DCR graph, and we additionally define OC-DCR choreographies based on this model. We call this extension Object-Centric Dynamic Condition Response Graphs (OC-DCR graphs).

[1] See http://dcrsolutions.net for tools available freely for academic use.

2. Building on the OC-DCR model, we define the formal model of DACEO Graphs as asynchronous Choreographies with asynchronous interactions between participants having message channels and the possibility of specifying high-level data-dependent ordering of messages.
3. We show that one can give a semantic preserving encoding of DACEO Graphs as synchronous OC-DCR Choreographies.

The remainder of the paper is structured as follows. In Sect. 2, we introduce the necessary preliminaries. We begin by presenting a DCR choreography example, which serves as a running example. We then formally define DCR graphs with nested groups, as well as choreographies based on this model. Section 3 introduces the two main contributions of this work: first, we define OC-DCR graphs and choreographies; second, we demonstrate how asynchronous communication can be modeled within this framework. Section 4 concludes and outlines promising directions for the further development of our modeling approach, as well as the limitations of the current solution.

2 Preliminaries

Below we recall mixed DCR Choreographies with data introduced in [13], with small changes in the presentation. Figure 1 shows a mixed DCR choreography for our running example. The boxes denote either groups of activities (dotted border) or activities (solid or dashed border). All activities have a participant role illustrated at the top, which indicates the role that can execute the activity. In this example, we have two roles *ProductService* and *CartService*. Activities can have different functionalities, and therefore can either be input activities, internal activities, data activities, or interactions. An input activity is an activity that receives a value from the environment when it is executed (denoted by?). Internal activities are without a specialized functionality, and data activities (denoted by a papercut) are considered data containers without the possibility to be executed. Interactions represent a message exchanged from the executing participant role to a receiving participant role, which is depicted with white text on black background at the bottom of the box.

The example reflects the slightly simplified semantics of the update price request within a microservice environment presented in Fig. 1. The product entries of the *Product Service* and the replicated product entries of the *Cart Service* are modeled here as data containers *Product* and *Replicated product*. In addition, the input activity *Input Price* offers the possibility to enter the parameters of the product whose price is to be changed via the environment. *Update Price* is an interaction between the two services, where *Product Service* is the sender and *Cart Service* is the receiver, and finally, on the *Cart Service* side, the *Update Replicated Price* activity receives the data and changes the price of the desired replicated product. The other elements of the graph, such as the colored relationships and conditional expressions, are now explained in detail step by step, accompanied by the formal definitions in the next section.

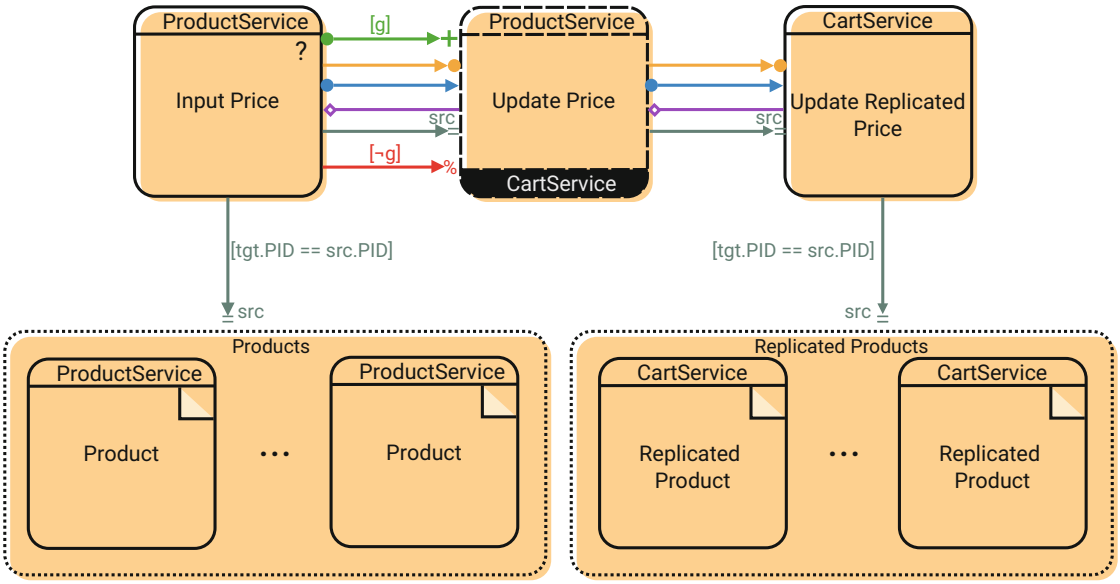


Fig. 2. Mixed DCR choreography with participant roles *Product Service* and *Cart Service* showing an interaction between the two services to update a product price, and the corresponding replicated product price of a specific product. Guard g is defined as $g = (\exists \alpha \in \mathcal{A}; l(\alpha).act = Product$ and $\alpha.PID == src.PID$ and $\alpha.Price \neq src.Price)$. Intuitively, guard g encodes whether the stored product price differs from the input product price for a matching product activity.

2.1 DCR Choreographies with Data and Nested Groups of Actions

We assume a fixed set of action names Act, ranged over by a, b, c, and a fixed set of participant roles Roles, ranged over by $r, r', r_1, r_2, \ldots$. We then define interactions and actions as follows.

Definition 1. *An* interaction *label is written as* $(a, r \to r')$*, in which the action* $a \in \mathsf{Act}$ *is* initiated *by the role* $r \in \mathsf{Roles}$ *and* received *by the role* $r' \in \mathsf{Roles} \backslash \{r\}$*, i.e., a role distinct from* r*. We denote by* $\mathsf{Int}_{\mathsf{Act}}$ *the set of all interactions over action names* Act*. A* input*,* local*, and* data *activity with the name* a *for role* r *is written as* $(a, ?r)$*,* (a, r) *and* $(a, @r)$ *respectively. We denote by* $\mathsf{Inp}_{\mathsf{Act}}$*,* $\mathsf{Loc}_{\mathsf{Act}}$ *and* $\mathsf{Data}_{\mathsf{Act}}$ *the sets of all input, local and data activities over action names* Act *respectively.*

The nodes of a DCR graph is a set of activities $\mathcal{A}$ labelled with an action name belonging to a set of action labels Act. Each activity is assigned a value by a value map $\mathsf{Va} : \mathcal{A} \to V$ from the set of activities to a set of typed values V (including the null value $\perp$ for all types). We refer to the type of $\mathsf{Va}(\alpha)$ as the type of the activity α. The types include basic types such as integers and strings and we also assume record types. For that we assume an infinite set of keys $\mathbb{K}$ and we further assume that $\mathcal{A} \subset \mathbb{K}$, i.e. the activity ids can be used as keys. Formally, a record can be defined as a map $[K \to V]$ for any finite subset of $K \subset \mathbb{K}$. We will write such a record as a set of key-value pairs: $\bar{p} = \{(k_1, v_1), \ldots, (k_n, v_n)\}$, where $k_i \in K$.

Given a graph G with activities $\mathcal{A}$ we let $\mathsf{Exp}_{\mathcal{A}}$ denote a set of expressions evaluating to typed values in V. We let $\mathsf{BExp}_{\mathcal{A}} \subseteq \mathsf{Exp}_{\mathcal{A}}$ denote the subset of boolean expressions and let $\mathsf{RExp}_{[\mathcal{A}_r],\mathcal{A}\bar{e}_i} \subseteq \mathsf{Exp}_{\mathcal{A}}$ denote the set of record expressions of the form $\{(\alpha_1, e_1), \ldots, (\alpha_n, e_n)\}$ where $\alpha_i \in \mathcal{A}_r$ and $e_i \in \mathsf{Exp}_{\mathcal{A}}$ is an expression matching the type of α_i.

For each $\alpha \in \mathcal{A}$ we assume an atomic expression α that evaluates to the current value of the activity. Examples of composite expressions are $\alpha+1$, "$Hello$"$++\alpha'$ and $\{(PID, 42), (SID, 101), (Price, \alpha)\}$, where α and α' are activities. The expression $\alpha + 1$ evaluates to the current value of α incremented with one, the expression "$Hello$" $++\alpha'$ evaluates to the string $Hello$ followed by the string contained in α' and finally, the expression $\{(PID, 42), (SID, 101), (Price, \alpha)\}$ evaluates to the record mapping PID to 42, SID to 101 and $Price$ to the current value of α. An exception is the usage of $l(\alpha)$, in which we refer to the label of the activity within the expression.

Expressions are in the present paper always associated with an edge of the graph. For a value map Va and an expression exp associated with an edge with source s and target t we write $[[exp]]_{\mathsf{Va},s,t}$ for the result of evaluating the expression exp. This allows us to introduce expressions src (source) and tgt (target) with evaluations $[[src]]_{\mathsf{Va},s,t} = \mathsf{Va}(s)$ and $[[tgt]]_{\mathsf{Va},s,t} = \mathsf{Va}(t)$. This is convenient when the same expression is used for a set of relations between different pairs of activities, which is possible using the notion of groups of activities and also using the concept of objects introduced in the following section. In our running example in Fig. 2, input activity *Input Price* expects a record containing three entries with keys *PID*,*SID*, and *Price*. The expression g is then used within an edge of the graph, to determine, whether an activity exists with action name *Product*, whose product id matches the product id of the Input Price activity (src), and whose price differs to the currently stored price. Intuitively, this expression checks, whether, for an incoming update price request, the desired product currently has a different price in the dataset.

We are now ready to formally define DCR graphs with data and nested groups.

Definition 2. *A* DCR Graph G with data and nested groups of activities *is given by a tuple* $(\mathcal{A}, \mathcal{A}^i, \mathcal{A}^p, \mathcal{G}, \triangleright, R, E, L, l, M)$ *where*

1. $\mathcal{A}$ *is a set of* activities *ranged over by* α,
2. $\mathcal{A}^i \subseteq \mathcal{A}$ *is the* input *activities,*
3. $\mathcal{A}^p \subseteq \mathcal{A}$ *is the* parameter *activities,*
4. $\mathcal{A}^d \subseteq \mathcal{A}$ *is the* data *activities,*
5. $\mathcal{G}$ *is a set of* activity groups *ranged over by* γ,
6. $\triangleright \subseteq A \times \mathcal{G}$ *is a* grouping *relation, where* $a \triangleright \gamma$ *reads* a *is member of* γ,
7. $R = \{\xrightarrow{g}\bullet, \xrightarrow{g}\diamond, \bullet\!\!\xrightarrow{g}, \xrightarrow{g}+, \xrightarrow{g}\%, \xrightarrow{g}=_e\}$ *are the relation types, for* $g \in \mathsf{BExp}_{\mathcal{A}}$ *and* $e \in \mathsf{Exp}_{\mathcal{A}}$
8. $E \subseteq A \times R \times A$ *is the relations,*
9. L *is the set of labels,*
10. $l\colon \mathcal{A} \to L$ *is a labelling function between activities and labels,*

11. $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va}) \in \mathcal{P}(\mathcal{A}) \times \mathcal{P}(\mathcal{A}) \times \mathcal{P}(\mathcal{A}) \times [\mathcal{A} \to V]$ *is the* marking,

where $A = \mathcal{A} \uplus \mathcal{G}$ *is the disjoint union of activities and activity groups. We write* $>$ *for* $\rhd^+$ *(the transitive closure of* $\rhd$*) and require that it is irreflexive. We write* $\geq$ *for reflexive closure of* $>$ *and* $\leq$ *for the inverse of* $\geq$*. We write* $a \phi a'$ *for* $\phi \in R$*, if* $(a'', \phi, a''') \in E.a \geq a'' \wedge a''' \leq a'$*.*

The initial state of G s defined by the marking M, where the last component Va is the value map assigning values to the activities and the first three will be explained further below. The parameter activities allow us to replace the value assigned to these activities. Given a value of record type $\bar{p} \in [\mathcal{A}^p \rightharpoonup V]$ that assign values to a subset of the parameter activities in $\mathcal{A}_p$ we let the graph $G[\bar{p}]$ denote the graph G with the marking $M[\bar{p}] = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va}[\bar{p}])$, where $\mathsf{Va}[\bar{p}](\alpha) = v$ if $\bar{p}(\alpha) = v$ and otherwise $\mathsf{Va}[p](\alpha) = \mathsf{Va}(\alpha)$.

The nested groups of activities provide a way to define relations to and from groups of activities, e.g., a relation (a, ϕ, a') where $a' \in \mathcal{G}$ means that there is a relation from a of type ϕ to all activities a'' nested in the group a', i.e., where $a'' > a'$. The boolean guard g on a rule relation means that the relation is to be disregarded, if g evaluates to false in the current marking M of the graph. In our running example in Fig. 2, the $\rightarrowtail$ relation from *Input Price* to the group *Products* and from *Update Replicated Price* to group *Replicated Products* is used to assign this relation to each of the *Product* and *Replicated Product* data activities inside the groups.

Now, a mixed DCR Choreography with data is simply a DCR Graph with data and groups of activies, where the activities are labelled with interactions, local input actions and internal actions.

Definition 3. *A triple* (G, A, R) *is a* Mixed DCR Choreography *when* G *is a DCR graph with data and nested groups of activities and labels* $L \subseteq \mathsf{Int}_{\mathsf{Act}} \cup \mathsf{Inp}_{\mathsf{Act}} \cup \mathsf{Loc}_{\mathsf{Act}}$*, and for any activity* α*, it holds that* $\alpha \in \mathcal{A}^i$ *if and only if* $l(\alpha) = ?(a, r)$ *for some* a *and* r*.*

2.2 Execution Semantics

Whether an activity α is enabled for execution depends on the marking M and the condition and milestone rules pointing to α. Intuitively, the set Ex contains the previously executed activities, the set Re of pending responses denotes the activities that must eventually be executed (or excluded) for the execution sequence to be completed and the set In denotes the activities that currently are included in the process. Only included activities may be executed and only included activities are considered in the relations. If $\alpha \rightarrow\!\bullet\, \alpha'$ we say α is a *condition* for α' and it denotes that the activity α' can only be executed if α has previously been executed, i.e. $\alpha \in \mathsf{Ex}$, or a is not included, i.e., $\alpha \notin \mathsf{In}$ If $\alpha \rightarrow\!\diamond\, \alpha'$ we say α is a *milestone* for α' and it denotes that the activity α' can only be executed if α is not a pending response or not included, i.e., $\alpha \notin \mathsf{Re} \cap \mathsf{In}$.

We formally define when an event is enabled as follows.

Definition 4. *Let G be a DCR graph with data and groups of activities, having the set of activities $\mathcal{A}$ and marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$. An activity α is* enabled*, which we will write* $\mathsf{enabled}(G, \alpha)$*, iff:*

1. $\alpha \in \mathsf{In} \backslash \mathcal{A}^d$,
2. $\forall \alpha' \in \mathsf{In}.\ \alpha' \xrightarrow{g}\bullet \alpha \wedge [[g]]_{M,\alpha',\alpha} \implies \alpha' \in \mathsf{Ex}$,
3. $\forall \alpha' \in \mathsf{In}.\ \alpha' \xrightarrow{g}\diamond \alpha \wedge [[g]]_{M,\alpha',\alpha} \implies \alpha' \notin \mathsf{Re}$.

When an enabled activity α in a graph G with marking M is executed, the marking is changed according to the current marking and the response, include, exclude and set value relations having the activity α as source. If $\alpha \in \mathcal{A}^i$, i.e. it is an input activity, the result also depends on an input value v provided by the environment. We let $\mathsf{Execute}(M, \alpha)$ denote the new marking if $\alpha \notin \mathcal{A}^i$ and let $\mathsf{Execute}(M, \alpha, v)$ denote the new marking if $\alpha \in \mathcal{A}^i$.

Definition 5. *Let $G = (\mathcal{A}, \mathcal{A}^i, \mathcal{A}^p, \mathcal{A}^d, \mathcal{G}, \triangleright, R, E, L, l, M)$ be a DCR graph with data and groups of activities, with $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$. For a value v, let $\mathsf{Execute}(M, \alpha, v) = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}', \mathsf{Va}')$ for*

1. $\mathsf{Ex}' = \mathsf{Ex} \cup \{\alpha\}$
2. $\mathsf{Re}' = (\mathsf{Re} \backslash \{\alpha\}) \cup \{\alpha' \mid \alpha \xrightarrow{g}\bullet \alpha' \wedge [[g]]_{M_v,\alpha,\alpha'}\}$
3. $\mathsf{In}' = (\mathsf{In} \backslash \{\alpha' \mid \alpha \xrightarrow{g}\% \alpha' \wedge [[g]]_{M_v,\alpha,\alpha'}\}) \cup \{\alpha' \mid \alpha \xrightarrow{g}+ \alpha' \wedge [[g]]_{M_v,\alpha,\alpha'}\}$
4. $\mathsf{Va}'(\alpha') = [[exp]]_{M_v,\alpha,\alpha'}$ *if* $\alpha \xrightarrow{g}\!\!= exp \wedge [[g]]_{M_v,\alpha,\alpha'}$
5. $\mathsf{Va}'(\alpha') = \mathsf{Va}_v(\alpha')$ *if not* $\alpha \xrightarrow{g}\!\!= exp\, \alpha' \wedge [[g]]_{M_v,\alpha,\alpha'}$
6. $M_v = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va}_v)$ *for* $\mathsf{Va}_v(\alpha) = v$ *and* $\mathsf{Va}_v(\alpha') = \mathsf{Va}(\alpha)$ *for* $\alpha' \neq \alpha$.

The effect of executing α is then the graph $G' = (\mathcal{A}, \mathcal{A}^i, \mathcal{A}^p, \mathcal{A}^d \mathcal{G}, \triangleright, R, E, L, l, M')$, where $M' = \mathsf{Execute}(M, \alpha, v)$ if $\alpha \in \mathcal{A}^i$ and v is the input value provided by the environment, and $M' = \mathsf{Execute}(M, \alpha, \mathsf{Va}(\alpha))$ if $\alpha \notin \mathcal{A}^i$. In the first case we write $G \xrightarrow{\alpha, v} G'$ and in the latter we write $G \xrightarrow{\alpha} G'$.

If $\alpha \bullet\!\!\rightarrow \alpha'$ we say α' is a *response* to α and it denotes that if α is executed, then α' is included in the set *Re* of pending responses. If $\alpha \rightarrow + \alpha'$ we say α includes α' and it denotes that if α is executed, then α' is included in the set In. If $\alpha \rightarrow \% \alpha'$ we say α excludes α' and it denotes that if α is executed, then α' is removed from the set In (if there is both an include and exclude relation between two activities, then the inclusion takes precedence). If $\alpha \rightarrow\!\!=_e \alpha'$ we say α sets the value of α' and it denotes that if α is executed, then the expression e is computed and the resulting value is assigned to the activity α in the value map $Va : \mathcal{A} \rightarrow V$. Finally, when an activity α is executed, it is removed from the set Re, unless it is a response to itself, i.e., $\alpha \bullet\!\!\rightarrow \alpha$.

Our example in Fig. 2 shows the application of these relations within the DCR choreography. After the previously described record consisting of the keys *PID*, *SID* and *Price* of the activity *Input Price* has been given by the environment and the activity has been executed, the product price of the product that has the corresponding *PID* is changed using the $\overset{tgt.PID == src.PID}{\rightarrow\!\!=}_{src}$ relation.

The use of the $\overset{\neg g}{\rightarrow\!\!\%}$ and $\overset{g}{\rightarrow\!\!+}$ relations between *Input Price* and *Update Price* here expresses that we only execute the interaction between *ProductService* and *CartService*, if the price of a product has changed. If the new price does not differ from the price of the old product price, the *Update Price interaction* is excluded. It is important to note that the order in which the relations are executed is decisive here. The inclusion or exclusion of activities is executed first and then the values are set using the $\rightarrow\!\!=$ relation. Furthermore, in the case of inclusion, *Update Price* is a response ($\bullet\!\!\rightarrow$) to *Input Price* and *Input Price* is in turn a condition ($\rightarrow\!\!\bullet$) for Update Price. This guarantees both that *Input Price* must be executed before *Update Price* and that *Update Price* is set to the pending state after execution. Additionally we guarantee, that *Update Product Price* can not be executed again, before the interaction *Update Price* between the two services has been executed, and therefore is not pending anymore, by using the milestone ($\rightarrow\!\!\diamond$) relation. This example choreography shows how DCR graphs can be used to model event-based distributed processes with data, but also highlights two serious limitations. Firstly, the set of product activities in the choreography is fixed because there is no mechanism to dynamically add new activities to the graph during execution. Secondly, the interaction between *ProductService* and *CartService* is assumed to be synchronous. However, in real-world microservice environments, communication between services is typically asynchronous. Therefore, we will address these limitations in the next section by enabling the dynamic creation of new activities as objects and supporting asynchronous interactions with configurable ordering guarantees.

3 Declarative Asynchronous Choreographies with General Data-Dependent Event-Ordering and Objects

Below, we first define an extension of DCR Graphs in Sect. 3.1, called Object-centric DCR (OC-DCR) Graphs, allowing dynamic instantiation of new sub graphs with relations between them, referred to as objects. We then extend the DCR choreographies to asynchronous choreographies with general data-dependent event-ordering and objects in Sect. 3.2.

3.1 Object-Centric Dynamic Condition Response Graphs

An Object-centric DCR Graph is a DCR graph with data and nested groups of activities, extended with a new relation that allows for a dynamical extension of the graph by instantiating objects from a finite set C (of classes), that are themselves object centric DCR Graphs.

We assume an infinite set of class names **Class** and let c range over class names. We also assume that every activity of an OC-DCR Graph is of the form $a@id$, where id is a unique identifier for the object it belongs to, created when it is instantiated, or the identifier *top* for activities initially belonging to the top-level graph. We then formally define object centric DCR Graphs as follows.

Definition 6. *An Object-centric DCR (OC-DCR) Graph is a tuple* $(G_\top, \rightarrow*_\top, C, \phi)$*, where* $C \subset_{fin}$ **Class** *is a finite set of classes and* ϕ *is the class definition function, assigning a class graph* $(G_c, \rightarrow*_c)$ *to each class in* C*, where*

1. $G_\zeta = (\mathcal{A}_\zeta, \mathcal{A}^i_\zeta, \mathcal{A}^p_\zeta, \mathcal{A}^d_\zeta, \mathcal{G}_\zeta, \rhd_\zeta, R, E_\zeta, L_\zeta, l_\zeta, M_\zeta)$*, where* $\zeta \in C \uplus \{\top\}$ *are DCR graphs with data and nested groups of activities and* $A_\zeta \cap \mathcal{A}_{\zeta'} = \emptyset$ *for* $\zeta \neq \zeta'$
2. $\rightarrow*_\zeta \subseteq A_\zeta \times \mathsf{BExp}_{\mathcal{A}_\zeta} \times \bigcup_{c \in \mathbf{Class}} (\mathsf{RExp}_{[\mathcal{A}^p_c], \mathcal{A}_\zeta \bar{\mathsf{e}}_\mathsf{i}} \times \{c\})$ *for* $\zeta \in C \uplus \{\top\}$ *are the* instantiation *relations, where we write* $a \xrightarrow{g}*_c \bar{p}c'$ *for* $(a, g, (\bar{p}, c')) \in \rightarrow*_c$*,*
3. $\mathcal{G}_\top = \mathcal{A}_\forall \cup \mathcal{G}$*, where* $\mathcal{A}_\forall = \bigcup_{c \in C} \{a_\forall \mid a \in \mathcal{A}_c\}$

The DCR graph $G_\top$ defines the top level graph, which corresponds to the initial process. Each graph G_c for $c \in C$ defines the graph that will be merged into the top level graph when a new object of class c is instantiated.

Only activities in the top level graph $G_\top$ of and OC-DCR Graph can be executed, and the enabledness is defined exactly as for DCR graphs with data and nested groups of activities. When it comes to defining the results of executing activities, we need to take into account instantiation of new objects. The meaning of the instantiation relation $\alpha \rightarrow*_c \bar{p}c'$ is that activity α in the graph G_c instantiates a fresh object defined by the class c' with parameters $\bar{p}$, i.e. a graph $G_{c'}[\bar{v}]$ is merged into the current graph, where $\bar{v}$ is the evaluation of $\bar{p}$.

We formalize the execution of the instantiation as follows. To make the presentation simpler, we assume that any activity α instantiates at most one object. The general case would just merge in the union of all the object graphs.

Definition 7. *Let* $G_{oc} = (G, \rightarrow *, C, \phi)$ *be an OC-DCR graph with marking* $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$ *and class graphs* $(G_c, \rightarrow *_c)$ *for* $c \in C$*. For* $\alpha \in \mathcal{A}$ *an enabled activity of* G *such that* $\alpha \xrightarrow{g}* \bar{p}c$ *and* $[[g]]_{M,\alpha,c} = \mathtt{tt}$ *and* $\bar{v} = [[\bar{e}]]_{M,\alpha,c}$ *is the value obtained by evaluating the parameter expression* $\bar{p}$ *in the marking* M*, let* $G_\# = (\mathcal{A}_\#, \mathcal{A}^i_\#, \mathcal{A}^p_\#, \mathcal{A}^d_\#, \mathcal{G}_\#, \rhd_\#, R, E_\#, L_\#, l_\#, M_\#)$ *denote the OC-DCR graph obtained from* $G_c[\bar{v}]$ *by replacing each activity or activity group* a *of* $G_c[\bar{v}]$ *with* $a@id$ *for* id *a fresh object identifier.*

The effect of executing α *is then the OC-DCR Graph* $G'_{oc} = (G', \rightarrow *, C, \phi)$*, where* $G' =$

$$(\mathcal{A} \cup \mathcal{A}_\#, \mathcal{A}^i \cup \mathcal{A}^i_\#, \mathcal{A}^p \cup \mathcal{A}^p_\#, \mathcal{A}^d \cup \mathcal{A}^d_\#, \mathcal{G} \cup \mathcal{G}_\#, \rhd' \cup \rhd_\#, R, E \cup E_\#, L \cup L_\#, l \cup l_\#, M'),$$

for $\rhd' = \rhd \cup \{(a, a_\forall \mid a \in \mathcal{A}_\#\}$ *and* $M' = \mathsf{Execute}(M \cup M_\#, \alpha, v)$ *if* $\alpha \in \mathcal{A}^i_\top$ *(where* Execute *is the function defined in Definition 5) and* v *is the input value provided by the environment and* $M' = \mathsf{Execute}(M \cup M_\#, \alpha)$ *if* $\alpha \notin \mathcal{A}^i_\top$*. In the first case we write* $G_{oc} \xrightarrow{\alpha, v} G'_{oc}$ *and in the latter we write* $G_{oc} \xrightarrow{\alpha} G'_{oc}$*.*

Note that the execution semantics defined above includes the execution semantics for activities that do not instantiate new objects as defined for DCR graphs with data and nested groups of activities, by updating the marking using the function defined in Definition 5. That is, if no objects are spawned only the marking is updated according to the effects of the other relations. for DCR

graphs with data and nested groups of activities, by updating the marking using the function Execute defined in Definition 5. That is, if no objects are spawned only the marking is updated according to the effects of the other relations.

Definition 8. *Let* $G_{oc} = (G, \rightarrow *, C, \phi)$ *be an OC-DCR graph with marking* $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$ *and class graphs* $(G_c, \rightarrow *_c)$ *for* $c \in C$*. For* $\alpha \in \mathcal{A}$ *an activity of* G *such* that it is not the case *that* $\alpha \xrightarrow{g} * \bar{p}c$ *and* $[[g]]_{M,\alpha,c} = \mathtt{tt}$*, then the effect of executing an enabled activity* α *is the OC-DCR Graph* $G'_{oc} = (G', \rightarrow *, C, \phi)$*, if* $G \xrightarrow{\psi} G'$ *for* $\psi = (\alpha, v)$ *or* $\psi = \alpha$*, which we write* $G_{oc} \xrightarrow{\psi} G'_{oc}$*.*

Definition 9. *A triple* (G_{oc}, A, R) *is a* Mixed OC-DCR Choreography *when* G *is an OC-DCR graph with both top level and class graphs having labels being subsets of* $\mathsf{Int}_{\mathsf{Act}} \cup \mathsf{Inp}_{\mathsf{Act}} \cup \mathsf{Loc}_{\mathsf{Act}}$*, and for any activity* α*, it holds that* $\alpha \in \mathcal{A}^i_\zeta$ *if and only if* $l(\alpha) = ?(a, r)$ *for some* a *and* r*.*

3.2 Asynchronous Choreographies with Data-Dependent Event-Ordering

We now introduce asynchronous interactions with general data-dependent ordering constraints, written as labels $(a, r \rightsquigarrow r', \mathsf{ord})$. We assume activities that are asynchronous interactions always have a record data type $[\mathsf{K} \rightarrow \mathsf{V}]$. The value of the activity at the time of execution of the activity by r is sent as a message that is eventually received by r'. The extra component ord in the label is an ordering constraint, which is a subset $\mathsf{ord} \subseteq K$ of the keys in the record type of the messages. The meaning is that any two messages m_1 and m_2 that agree on the values of the keys in ord, i.e. $\forall k \in \mathsf{ord}. m_1.k = m_2.k$, will be received in the order they are sent. In particular, this means that the empty ordering constraint $\emptyset$ declare a FIFO ordering of messages, since every message is received in the order sent, while the ordering constraint K essentially corresponds to declaring an unordered communication, since only fully identical messages will be received in order (and thus might as well have been out of order). In between the two extremes, we can order messages by a non-empty but true subset of keys, e.g. in the running example for the update price interaction we can define that only messages pertaining to the same product will be received in the order they are sent by using the ordering constraint $\{PID\}$. Finally, we introduce for each asynchronous interaction activity α a group of activities ®α. The intuition is that this group represents the reception of asynchronous messages. We can then express rules constraining or reacting to the reception of a message by adding relations to an from these groups.

We formally define the notation of asynchronous interactions with data-dependent ordering as follows.

Definition 10. *An* asynchronous interaction *with data-dependent ordering is a tuple* $(a, r \rightsquigarrow r', \mathsf{ord})$*, where* $a \in \mathsf{Act}$ *is the action name,* $\mathsf{ord} \subset_{fin} \mathbb{K}$ *is the ordering constraint,* r *is the sending participant role and* r' *is the receiving participant role and* $r \neq r'$*. We denote by* $\mathsf{AInt}_{\mathsf{Act}}$ *the set of all asynchronous interactions over action names* Act*.*

We can now define our model DACEO, of Declarative Asynchronous Choreographies with general data-dependent Event-Ordering, as DCR Choreographies over OC-DCR Graphs and asynchronous interactions and local actions and data actions as labels.

Definition 11. *A triple* (G, A, R, P) *is a* Declarative Asynchronous Choreography with Data-dependent ordering (DACEO) Graph *when* G *is an OC-DCR graph with labels* $L \subseteq \mathsf{AInt}_{\mathsf{Act}} \cup \mathsf{Inp}_{\mathsf{Act}} \cup \mathsf{Loc}_{\mathsf{Act}}$*, for some set of activity names* A *and for any activity* α*, it holds that* $\alpha \in \mathcal{A}^i$ *if and only if* $l(\alpha) = ?(a, r)$ *for some* a *and* r*. We also require that there are groups of activities* $\{®\alpha \mid l(\alpha) \in \mathsf{AInt}_{\mathsf{Act}}\} \subseteq \mathcal{G}$*, i.e. there is an activity group for each asynchronous interaction activity. Moreover, there can be no condition or milestone relations having a group* $®\alpha$ *as source, nor can there be any response, include, exclude or setvalue relation having a group* $®\alpha$ *as target. Finally, if* $l(\alpha) = (a, r \rightsquigarrow r', \mathsf{ord})$ *then the type of* α *is a record type* $[\mathsf{K} \to \mathsf{V}]$ *and* $\mathsf{ord} \subseteq K$ *and* $P(\alpha) = (P_\alpha, n_\alpha)$*, where* $n_\alpha \geq 0$ *is an integer (representing the number of messages sent),* P_α *is a set of pairs* $(\bar{v}, i)$*, where* $i \in [1 \ldots n_\alpha]$ *denote the message number and* $\bar{v} \in [\mathsf{K} \to \mathsf{V}]$ *is a record value (of the send message) that is waiting to be received, such that* $(\bar{v}, i), (\bar{v}', i) \in P_\alpha$ *implies* $\bar{v} = \bar{v}'$*, i.e. the message numbers are unique.*

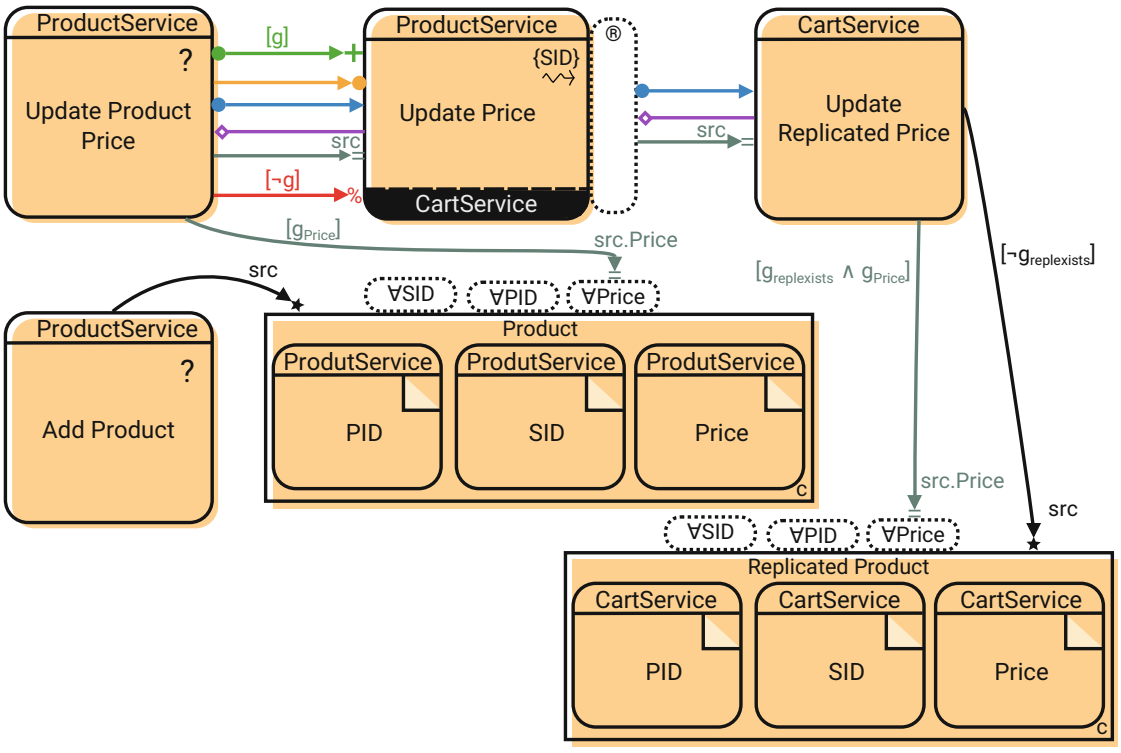


Fig. 3. A DACEO Graph expressing asynchronous update price request between *Product Service* and *Cart Service*.

Figure 3 shows an extension of the DCR graph from Fig. 2 with the elements introduced in this section. The relationships between *Update Product Price* and

Update Price, as well as *Update Price* and *Update Replicated Price* are already known from the previous example. The following two things have fundamentally changed in the choreography.

Firstly, instead of having a fixed set of products and replicated products, we use objects with class ids *Product* and *Replicated Product*, illustrated with a small c, for allowing dynamic spawning of these objects. Both classes have three members *PID*, *SID* and *Price* in this example, which are represented in the illustration by showing the data activities, together with the corresponding ∀-groups. When a new product is spawned, the three members *SID*, *PID* and *Price* are instantiated accordingly with the values of the record value in the *Create Product* activity. The record is directly accessible by using the *src* keyword and can be passed for instantiation. For example, if activity *Create Product* is executed with record $p = \{(SID, 1), (PID, 101), (Price, 1000)\}$, then p can be accessed in the spawn relation with the *src*-keyword as instantiation, and the three activities *SID* containing value 1, *PID* containing value 101, and *Price* containing value 1000, are sorted into the three ∀-groups together with a fresh unique object id. The guard g_{price} is used here to find the price activity that is linked to the desired product to be changed.

$$g_{price} = \exists \alpha \in \mathcal{A}.l(\alpha) == PID \wedge \alpha == src.PID \wedge \alpha.oid == src.oid$$

It expresses that we only change the value of the price for the *Price* activity that is assigned to the correct *PID* activity, using the object identifier (*oid*) to link the two activities together.

Secondly, the previously synchronous interaction *Update Price* from Fig. 2 is now an asynchronous interaction. The order of the messages to be sent is displayed graphically in the top right-hand corner. In our example, the order of incoming messages must be maintained according to their causality in relation to the seller identification. The group labelled ® next to the asynchronous interaction is a container that graphically contains the messages that have been sent but not yet received. We show an example of this in the next section.

Last, the *Update Replicated Price* activity activates the spawning of a new replicated product if the product whose price is to be changed is not yet contained in the set of replicated product objects. This occurs when the price of a product is changed for the first time, as the addition of a new product within the *ProductService* does not automatically lead to an addition in the *CartService*. If the product already exists, the price of the desired replicated product is changed in the same way as before int the *ProductService*. The guard $g_{replexists}$ is used here as a negated guard to check for non existence of the replicated product on the $\rightarrow *$ relation and in on the $\twoheadrightarrow$ relation to verify the existence of the replicated product.

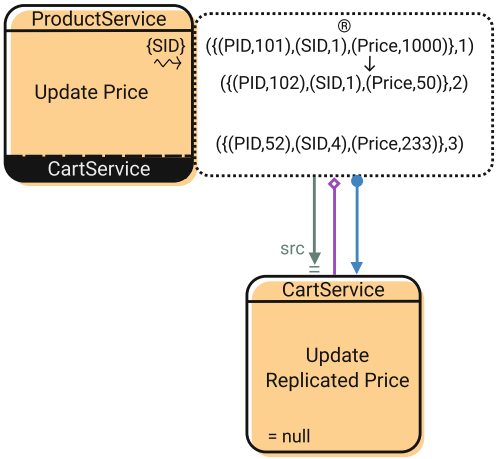
$$g_{replexists} = \exists \alpha \in \mathcal{A}.l(\alpha) == PID \wedge \alpha == src.PID$$

3.3 Semantics

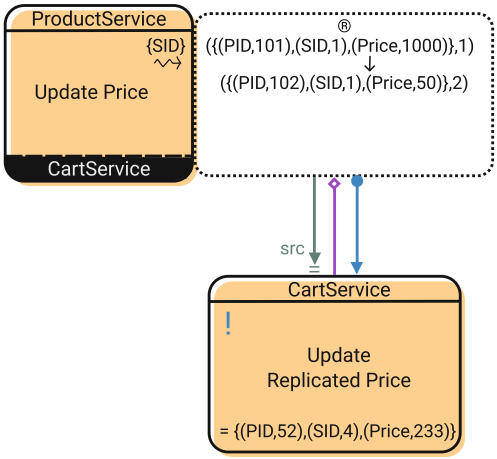
The semantic of DACEO Graphs extends the semantics of mixed OC-DCR Choreographies to take into account the messages waiting to be received. This is done by considering the graph obtained by extending the graph with pending activities corresponding to the messages pending to be received. Figure 4 shows a graphical representation of a part of the DACEO Graph for the running example consisting of only the *Update Price* asynchronous interaction and the local *Update Replicated Price* activity of the Cart Service. The sub Fig. 4a

to the left shows the state of the message channel after the sending of three messages and the order that can be derived from the order constraint $\{SID\}$ is shown by an arrow from message number 1 to message number 2. The sub Fig. 4b

to the right shows the updated state after the execution of the reception of message number 3. Note that due to the response relation $\bullet\!\!\rightarrow$, the local activity *Update Replicated Price* is now pending (indicated with an exclamation mark in the top left corner), i.e. it is in the Re set of the marking. The milestone relation $\rightarrow\!\!\diamond$ will now block the reception of further messages until *Update Replicated Price* has been executed. Finally, due to the $\rightarrow\!\!=$ relation, the value of the local activity *Update Replicated Price* has been set to the value of the message.



(a) State of execution of an excerpt of the full DACEO Graph in figure 3 showing three messages being asynchronously send.



(b) State of execution of an excerpt of the full DACEO Graph in figure 3 showing the reception of message with number 3. The value of *Update Replicated Price* has been set to the content of the message.

Fig. 4. .

Definition 12. *Let (G, A, R, P) be a DACEO Graph having the set of activities $\mathcal{A}$ and marking M. Let the graph $G^{®}$ be the graph obtained from G by extending*

the activities to $\mathcal{A}^{®} = \mathcal{A} \cup \mathbb{P}$*, the grouping relation to* $\triangleright^{®} = \triangleright \cup \{((\alpha, m), ®\alpha) \mid (\alpha, m) \in \mathbb{P}\}$ *and extending the marking to* $M^{®} = (\mathsf{Ex}, \mathsf{Re} \cup \mathbb{P}, \mathsf{In} \cup \mathbb{P}, \mathsf{Va}^{®})$*, where* $\mathbb{P} = \bigcup_{\alpha \in \mathcal{A}} \{(\alpha, m) \mid m \in P(\alpha)\}$ *is the union of all the message sets defined by* P *(guaranteed to be disjoint by adding the activity id to each message) and* $\mathsf{Va}^{®}(\alpha) = \bar{v}$*, if* $\alpha = (\alpha', (\bar{v}, j)) \in \mathbb{P}$ *and* $\mathsf{Va}^{®}(\alpha) = \mathsf{Va}(\alpha)$ *for* $\alpha \in \mathcal{A}$*.*

We can now define when activities and message reception is enabled by using the standard definition of enabledness for the graph extended with explicit messages and taking the ordering constraint on messages into account.

Definition 13. *Let* $G_{aoc} = (G, A, R, P)$ *be a DACEO Graph having the set of activities* $\mathcal{A}$ *and marking* M*. An activity or reception of a message* $\alpha \in \mathcal{A} \cup \mathbb{P}$ *is* enabled *in* G_{aoc}*, which we write* $\mathsf{enabled}(G_{aoc}, \alpha)$*, if and only if* $\mathsf{enabled}(G^{®}, \alpha)$ *and if* $\alpha = (\alpha', (\bar{v}, i))$ *and* $l(\alpha') = (a, r \rightsquigarrow r', \mathsf{ord})$ *then* $\forall (\bar{v}', i') \in P(\alpha').((\forall k \in \mathsf{ord}.\bar{v}.k = \bar{v}'.k) \implies i \leq i')$*.*

The execution of asynchronous interactions α is extended to also adding a new message to the set $P(\alpha)$ of messages waiting to be received.

Definition 14. *Let* $G_{aoc} = (G, A, R, P)$ *be a DACEO Graph having the set of activities* $\mathcal{A}$ *and marking* $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$*. If* $\mathsf{enabled}(G_{aoc}, \alpha)$ *and* $l(\alpha) = (a, r \rightsquigarrow r', \mathsf{ord})$ *then the effect of executing* α *is* $G'_{occ} = (G', A, R, P')$ *if* $G \xrightarrow{\alpha} G'$ *and* $P'(\alpha) = (P_\alpha \cup \{(\mathsf{Va}(\alpha), n)\}, n + 1)$ *if* $P(\alpha) = (P_\alpha, n)$ *and* $P'(\alpha') = P(\alpha')$ *for* $\alpha' \neq \alpha$*.*

Dually, the execution (i.e. reception) of an enabled message m in $P(\alpha)$ waiting to be received removes the message from the set $P(\alpha)$, formally defined as follows.

Definition 15. *Let* $G_{aoc} = (G, A, R, P)$ *be a DACEO Graph. If* $\mathsf{enabled}(G, (\alpha, m))$ *for* $(\alpha, m) \in \mathbb{P}$ *then the effect of executing the reception of* (α, m) *is* $G'_{occ} = (G', A, R, P')$ *if* $G^{®} \xrightarrow{(\alpha,m)} G''$ *and* $P'(\alpha) = (P_\alpha \backslash \{m\}, n)$ *if* $P(\alpha) = (P_\alpha, n)$ *and* $P'(\alpha') = P(\alpha')$ *for* $\alpha' \neq \alpha$*, and* G' *is the graph obtained from* G'' *by removing all the explicit message activities from the set of activities, the nesting relation and the marking.*

3.4 Encoding DACEO Graphs as OC-DCR Choreographies

We now describe how a DACEO Graph G_{aoc} can be encoded as a semantically equivalent OC-DCR Choreography $enc(G_{aoc})$. Figure 5 illustrates the encoding of the asynchronous interaction activity and the local activity in the CartService, shown in the state after three messages have been sent as illustrated in Fig. 4a.

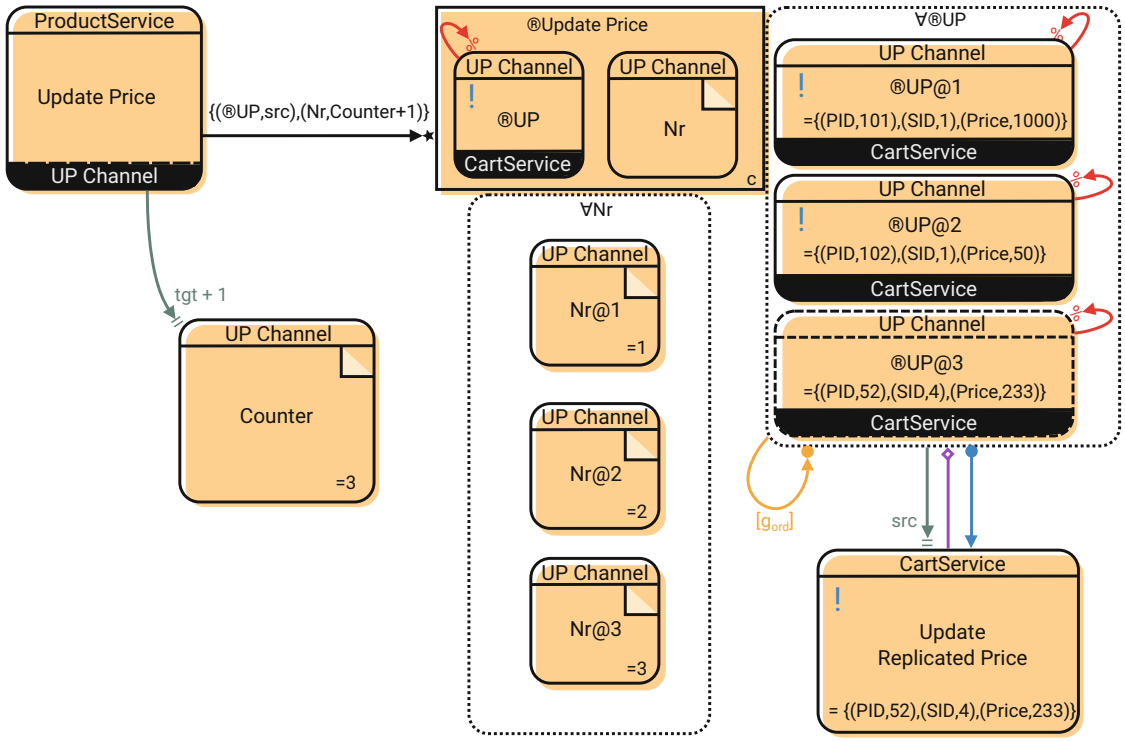


Fig. 5. Encoding of the example in Fig. 4 as OC-DCR Graph.

The idea of the encoding is to explicitly model the messages as objects instantiated at a channel. Firstly, for each asynchronous interaction activity α labelled $(a, r \rightsquigarrow r', \mathsf{ord})$ we introduce a message channel $\alpha\,\mathsf{Channel}$ as a new participant role and a class $®\alpha$ representing the messages. The class $®\alpha$ is defined as an OC-DCR Choreography containing one interaction activity $®\alpha$ with label $(®a, \alpha\,Channel \rightarrow r')$ (encoding the message delivery) and one data activity Nr with label $(Nr, @a\,Channel)$ (encoding the message numbering). The interaction activity $®\alpha$ has an exclude relation to itself (modelling that the message can only be received once) and is pending, i.e. it is in the response set of the marking of the class, modelling that the message must eventually be received. Finally, we model the message ordering by a condition relation from the activity group $\forall®\alpha$ to itself, which is guarded by an expression g_{ord} that encodes the ordering constraint. The expression consists of two parts $g_{\mathsf{ord}} = g'_{\mathsf{ord}} \wedge g_{\mathsf{sent}}$, where $g'_{\mathsf{ord}} = \forall k \in \mathsf{ord}.tgt.k = src.k$ encodes the messages that must be ordered, and the second part $g_{\mathsf{sent}} = \exists \alpha_s, \alpha_t.l(\alpha_s) = l(\alpha_t) = (Nr, @\alpha Channel) \wedge \alpha_s.oid = src.oid \wedge \alpha_t.oid = tgt.oid \wedge \alpha_s < \alpha_t$ encodes the order in which these messages were sent. (The expression $\exists \alpha_s, \alpha_t.l(\alpha_s) = l(\alpha_t) = (Nr, @\alpha Channel) \wedge \alpha_s.oid = src.oid \wedge \alpha_t.oid = tgt.oid$ encodes that α_s and α_t are the Nr activities in the objects of, respectively, the source and target message activities for the condition relation).

Then, we replace the asynchronous interaction activity by a synchronous interaction $(a, r \rightarrow \mathsf{aChannel})$ that instantiate a message object from the message class $®a$, containing the original message as an activity and a sequence number

data activity, in the Ether. The message activities are pending synchronous interactions from the Ether to the role r' that excludes themselves, i.e., they have an exclusion relation from themselves to themselves. Finally, the delivery ordering of the messages is encoded by a condition relation from all the messages of instantiated objects to themselves, guarded by the ordering constraint. In order to encode the increasing order numbering of messages, we introduce a counter data object with the role $\alpha\,Channel$ which initially contains the integer value 0. We then add a set value relation from the interaction activity α labelled $(a, r \rightsquigarrow r', \mathsf{ord})$ (encoding the sending of the message) to the counter data object, which has the expression $\mathsf{tgt} + 1$, i.e. it evaluates to the increment of the value of the target activity. We can then use the value of this data activity as a parameter when we instantiate a new message object.

The final step is to replace all the relations from the activity group $®\alpha$ in G_{aoc} with relations from the activity group $\forall®\alpha$ representing all the message activities in the class $®\alpha$.

We are now ready to state the semantic correspondence. In lack of space we only provide the proof idea.

Theorem 1. *For a DACEO Graph G_{aoc}, the OC-DCR Choreography $enc(G_{aoc})$ has equivalent transition semantics.*

Proof Idea: The correspondence follows by inspecting the semantics of DACEO and OC-DCR Grahps. In order to formally specify the correspondence, we first extend DACEO graphs with an extra set $RM(\alpha)$ of received messages for each asynchronous interaction activity α and extend Definition 15 such that the received messages are added to this set. We can now define the corresponding synchronous OC-DCR Choreography by the above mapping and mapping the received messages to message objects with excluded message data objects.

4 Conclusion and Future Work

In this paper, we introduced DACEO, a formal modeling language designed for declarative asynchronous choreographies with data-dependent message ordering and data objects. By equipping DACEO with an execution semantic, we enable effective modeling of asynchronous distributed event-based systems where total message delivery ordering cannot be assumed. Our model builds upon and extends the prior work on synchronous declarative choreography models based on DCR Graphs with object-centric Dynamic Condition Response Graphs, ensuring compatibility and semantic preservation. We demonstrated its practical applicability through a running example of an online marketplace, showcasing how it enforces causal consistency constraints in an asynchronous, event-based microservices architecture. This example highlights the potential of DACEO in real-world scenarios, providing a robust framework for managing complex data-dependent interactions. Overall, DACEO represents a significant advancement in the modeling of asynchronous choreographies.

In future work, we aim to explore the potential of endpoint projection, which, while not necessary for monitoring, can facilitate the creation of individual processes for each participant based on the global specification. This approach could enhance the flexibility and scalability of our model. Additionally, we plan to extend our framework to support more than binary message exchanges between participants, thereby enriching the interaction patterns. By incorporating asynchronous semantics into the executable DCR model, we can achieve an executable model that enforces event ordering, further strengthening the robustness and applicability of our approach in distributed systems.

References

1. Aalst, W.: Object-centric process mining: dealing with divergence and convergence in event data. In: Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724, pp. 3–25. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_1
2. Bailis, P., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Bolt-on causal consistency. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, pp. 761–772. Association for Computing Machinery, New York (2013). https://doi.org/10.1145/2463676.2465279
3. Barbanera, F., Lanese, I., Tuosto, E.: Choreography automata. In: Bliudze, S., Bocchi, L. (eds.) COORDINATION 2020. LNCS, vol. 12134, pp. 86–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50029-0_6
4. Calvanese, D., Montali, M., Patrizi, F., Rivkin, A.: Modeling and in-database management of relational, data-aware processes. In: Giorgini, P., Weber, B. (eds.) CAiSE 2019. LNCS, vol. 11483, pp. 328–345. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21290-2_21
5. Christfort, A., Rivkin, A., Fahland, D., Hildebrandt, T., Slaats, T.: Discovery of object-centric declarative models. In: Proceedings of 6th International Conference on Process Mining ICPM (2024)
6. Ciccio, C.D., Marrella, A., Russo, A.: Knowledge-intensive processes: characteristics, requirements and analysis of contemporary approaches. J. Data Semant. **4**, 29–57 (2015). https://api.semanticscholar.org/CorpusID:16082882
7. Costa Seco, J., Debois, S., Hildebrandt, T., Slaats, T.: Reseda: declaring live event-driven computations as reactive semi-structured data. In: 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), pp. 75–84 (2018). https://doi.org/10.1109/EDOC.2018.00020
8. Debois, S., Hildebrandt, T.T., Slaats, T.: Replication, refinement & reachability: complexity in dynamic condition-response graphs. Acta Informatica **55**(6), 489–520 (2017). https://doi.org/10.1007/s00236-017-0303-8
9. Dezani-Ciancaglini, M., Drossopoulou, S., Mostrous, D., Yoshida, N.: Objects and session types. Inf. Comput. **207**(5), 595–641 (2009). https://doi.org/10.1016/j.ic.2008.03.028. https://www.sciencedirect.com/science/article/pii/S0890540109000261
10. Ferreira Loff, J.a., Porto, D., Garcia, J.a., Mace, J., Rodrigues, R.: Antipode: enforcing cross-service causal consistency in distributed applications. In: Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23, pp. 298–313. Association for Computing Machinery, New York (2023). https://doi.org/10.1145/3600006.3613176

11. Gay, S.J., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular session types for distributed object-oriented programming. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '10, pp. 299–312. Association for Computing Machinery, New York (2010). https://doi.org/10.1145/1706299.1706335, https://doi.org/10.1145/1706299.1706335
12. Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Delta-BPMN: a concrete language and verifier for data-aware BPMN. In: Polyvyanyy, A., Wynn, M.T., Van Looy, A., Reichert, M. (eds.) BPM 2021. LNCS, vol. 12875, pp. 179–196. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85469-0_13
13. Hildebrandt, T.T., López, H.A., Slaats, T.: Declarative choreographies with time and data. In: Di Francescomarino, C., Burattin, A., Janiesch, C., Sadiq, S. (eds.) Business Process Management Forum, vol. 490, pp. 73–89. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-41623-1_5
14. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1) (2016). https://doi.org/10.1145/2827695
15. Laigner, R., Zhang, Z., Liu, Y., Gomes, L.F., Zhou, Y.: Online marketplace: a benchmark for data management in microservices. Proc. ACM Manag. Data **3**(1) (2025). https://doi.org/10.1145/3709653
16. Laigner, R., Zhou, Y., Salles, M.A.V., Liu, Y., Kalinowski, M.: Data management in microservices: state of the practice, challenges, and research directions. Proc. VLDB Endow. **14**(13), 3348–3361 (2021). https://doi.org/10.14778/3484224.3484232
17. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pp. 401–416. Association for Computing Machinery, New York (2011). https://doi.org/10.1145/2043556.2043593
18. Maggi, F.M., Marrella, A., Patrizi, F., Skydanienko, V.: Data-aware declarative process mining with sat. ACM Trans. Intell. Syst. Technol. **14**(4) (2023). https://doi.org/10.1145/3600106