Choreography Revisited*

Jianwen Su and Yutian Sun

Department of Computer Science, UC Santa Barbara, USA

Abstract. A choreography models interoperation among multiple participants in a distributed environment. Existing choreography specification languages focus mostly on message sequences and are weak in modeling data shared by participants and used in sequence constraints. They further assume a fixed number of participants and make no distinction between participant types and participant instances. Artifact-centric business process models give equal considerations on modeling data and on control flow of activities. These models provide a solid foundation for choreography specification. Through a detailed exploration of an example, this paper introduces a choreography language for artifacts that is able to specify data conditions and the instance-level correlations among participants.

1 Introduction

Modern enterprises rely on business process management systems to support their business, information flows, and data analytics [7]. Interoperation among business processes (BPs) (in a distributed environment) continues to be a fundamental challenge. In general, two approaches [9, 6], namely *orchestration* and *choreography*, are used to model interoperation. An orchestration requires a designated "mediator" to communicate with and coordinate all participating BPs. BPEL [1] is a typical orchestration language and has been widely used in practice. However, orchestration reduces the autonomy of participating BPs and does not scale well due to the mediator. The choreography approach specifies desirable global behaviors among participating BPs but otherwise leaves the BPs to operate autonomously and communicate in peer-to-peer fashion. One difficulty for this approach is to coordinate among participating BPs in absence of a central control point. This paper introduces a language for choreography specification through a detailed example. The formal model, syntax and semantics of this language were reported in [10].

A choreography models interoperations among multiple participants in a distributed environment. A choreography may be specified as a state machine representing message exchanges between two parties [5] or permissible messages sequences among two or more parties with FIFO queues [2]. It may even be specified as individual pieces using patterns [12], or implicitly through participants behaviors [4].

Data play an essential role in process modeling [8]. Interoperation of BPs also needs data to specify precisely global behaviors among participants. Existing choreography languages focus mostly on specifying message sequences and are weak in modeling data shared by participants and used in choreography constraints. A tightly integrated

* Supported in part by a grant from Bosch.

data model with message sequence constraints would allow a choreography to constrain execution accurately. More importantly, the existing choreography languages (with an exception of [10]) assume a fixed number of participants and make no distinction between *participant types* and *participant instances*. For example, an *Order* process instance may communicate with many correlated *Vendor* process instances. Therefore, a choreography language must be able to model correlations between process instances.

Artifact-centric process models [8] have attracted increasing attention in modeling BPs. An artifact model includes an information model for business data and a specification of lifecycle that defines permitted sequences of activities. Artifact models provide an excellent starting point for developing choreography specification.

This paper focuses on choreography specification with process instance correlations and data. Through a detailed running example, this paper demonstrates the following four aspects of the language: (1) Each participant type is an artifact model with a specified part of its *information model* accessible by choreography specification. (2) Correlations between participant types and *instances* are explicitly specified, along with *cardinality constraints* on correlated instances. (3) Messages can include data; both message data and artifact data can be used in specifying choreography constraints. (4) Our language is declarative and uses logic rules based on a mix of first-order logic and a set of binary operators from DecSerFlow [11]. In summary, this paper explains the details of the choreography model and language presented in [10] and can be viewed as a companion paper of [10].

The remainder of the paper is organized as follows. Section 2 introduces a collaborative BP example used throughout the paper. Sections 3 to 7 present the five main components of the choreography language respectively. Section 8 briefly discusses the semantics of the language. Section 9. concludes the paper.

2 A Running Example

To illustrate the choreography language, in this section we describe an online shopping example that involve different participants in a collaborative business process.

Consider an online *store* whose items are owned and provided by many *vendors*. A vendor may use several *warehouses* to store and manage its inventory. Once the customer completes shopping, she initiates a payment process in her *bank* that will send a check to the *store* on her behalf. Meanwhile, the *store* groups the items in her cart (1) by *warehouses* and sends to each *warehouse* for fulfillment and shipping, and (2) by *vendors* and subsequently requests each *vendor* to complete the purchase. The *vendors* inform *warehouses* upon completion of purchase. After the *store* receives the payment and *vendors*' completion of purchases, the *store* asks *warehouses* to proceed with shipping.

In this example, four types of participants (*store*, *vendor*, *warehouses*, and *bank*) are involved and each type has/can be viewed as its own business process. Although *store* and *bank* have only one process instance each for a single customer shopping session, there may be multiple instances for *vendors* as well as for *warehouses*. To design a choreography for such collaborative processes with multiple collaborating participants, the following five components are used in our language: *artifacts*, *correlation graphs*, *derived correlations*, *messages*, and *choreography constraints*.

Artifacts: In artifact-centric modeling, an artifact instance encapsulates a running process. For example, the *store* initiates an `Order` (artifact) instance that handles the processing of an customer order. The `Order` instance contains the needed business data such as items, customer names, date, etc. as well as a lifecycle that guides how the process should progress. Similarly, other participant processes are also artifact instances, `Purchase` instances represent order processing at *vendors*, `Fulfillment` instances are packing and delivery processes at *warehouses*, and a `Payment` instance is initiated upon a customer request to make a payment to the online *store*.

Correlation graphs: Over the (defined and) participating artifacts, a graph is used to specify correlations among these artifacts. Essentially, correlations represent the need for one process instance to communicate with another. For example, an edge between `Order` (artifact) and `Purchase` represents that one `Order` instance can correlate with many `Purchase` instances.

Derived Correlations: In addition the correlation specified by correlation graphs, there are some correlations that can only specified by "rules" rather than edges in the correlation graph. For example, a `Purchase` instance and a `Fulfillment` instance are correlated when they share the same item ordered by a customer. (Thus, they need to communicate, e.g., on the status of purchase processing by the *vendor*.) We call such rule-specified correlations "derived correlations".

Messages: Correlated participants can send messages between them. For example, a `Purchase` instance can inform the completion of the purchase to all correlated `Fulfillment` instances by sending 'PC' (purchase complete) messages; or, the customer is informed the completion of the order by receiving an 'OC' (order complete) message from the `Order` instance.

Choreography Constraints: Finally, a choreography is a specification of how, when, and what messages should be sent among participants. To achieve this, declarative constraints are used. An example choreography constraint can be that whenever a PC (purchase complete) message is sent, a SC (shipping complete) message is sent in the future.

We will introduce each of the above five components in Sections 3 to 7, respectively, and illustrate the semantics of the choreography language in Section 8.

3 Artifacts

In our model, artifacts represent participant BPs, the notion of an "artifact interface" captures an artifact with "visible" data contents for choreography specification.

An *artifact* (*interface*) is a complex data type. The attributes in an artifact can be accessed in choreography. Each artifact always contains a top-level and non-set-typed attribute 'ID' to hold a unique identifier for each artifact instance. The data type of an attribute is hierarchically organized.

Fig. 1 shows interfaces of the four artifacts mentioned in Section 2, namely, `Order`, `Payment`, `Purchase`, and `Fulfillment`. An `Order` artifact (interface) contains 'ID', order 'Info', 'Customer', and shopping 'Cart' as its top-level attributes, among which, 'Info', 'Customer', and 'Cart' are complex attributes that contain nested attributes. Moreover, 'Cart' is a relation-typed attribute (indicated by '⊛') that may include 0 or more tuples with six (nested) attributes: 'Inv(entory)_ID', 'Item' name, 'Quantity', 'Unit_Price', 'Vendor' (seller of the item), and 'Warehouse' (location of the item).

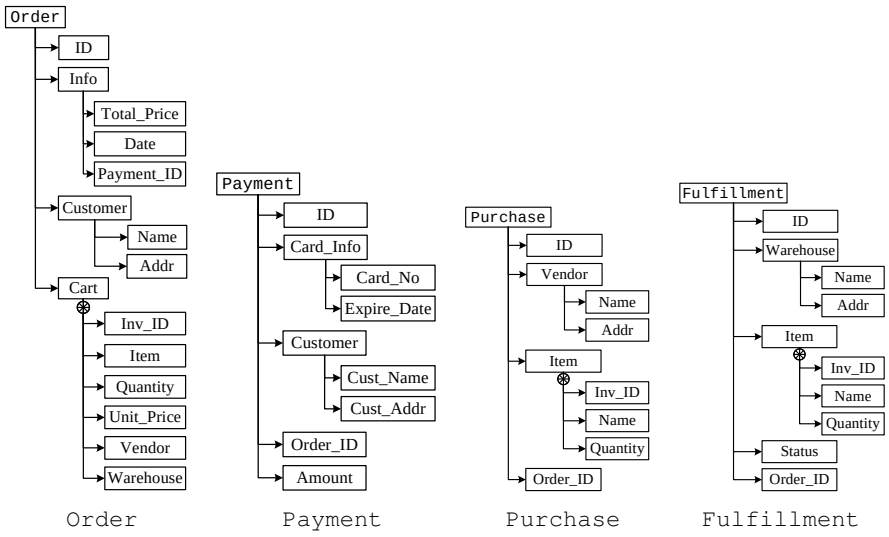

Fig. 1. artifact interfaces

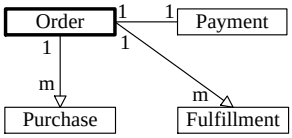

Fig. 2. Correlation graph

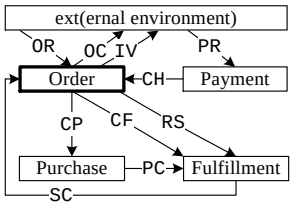

Fig. 3. Message diagram

The `Payment` interface contains attributes such as "Card_No" and "Cust(omer)_Name" to record the payment information. The `Purchase` and `Fulfillment` interfaces are structured similarly. Note that the "Item" attribute of `Purchase` and the "Item" attribute of `Fulfillment` have the same structure but in general may store different values. The "Item" of `Purchase` stores all items purchases by a customer that are provided by the same vendor; while the "Item" of `Fulfillment` stores all items purchases by a customer that are stored in the same warehouse.

The attributes in an artifact interface can be accessed in choreography. Each artifact interface always contains the attribute 'ID' to hold a unique identifier for each artifact instance. The data type of an attribute can be hierarchical or another artifact interface; in the latter case, values of the attribute are identifiers of the referenced interface.

Given an artifact interface, an *artifact instance* is a partial mapping from all the attributes of the artifact to their corresponding domains, such that ID is defined and unique.

4 Correlation Graphs

We now introduce an important notion of a "correlation graph". Intuitively, a correlation graph specifies whether two BP instances (or equivalently, artifact instances) are correlated and whether the correlation is one instance of a BP correlating to 1 or many instances of the other BP. Similar to WS-CDL, only a pair of correlated instances may exchange messages in our model.

Informally, a *correlation graph* is a rooted, acyclic, and connected graph whose nodes represent artifacts and edges denote "correlations" among the artifacts. More precisely, if two artifacts are correlated (connected by an edge), it indicates that some instances of these two artifact interfaces are correlated. For example, Fig. 2 shows the correlation among the four artifacts in Fig. 1. The root of this graph is the `Order` artifact, which means that the creation of an `Order` artifact instance also starts the entire collaborative process instance. Our model restricts that there exists exactly one root artifact instance in a collaborative process instance. The three edges in Fig. 2 indicate that some instances of `Order` are correlated with some instances of `Payment`, `Purchase`, and `Fulfillment` respectively.

The edges among nodes (i.e., artifacts) can be *directed* to denote a creation relationship (correlation will be created during execution through messages) or *undirected* to denote a correlation that is externally set up. Moreover, each edge should have a *cardinality*, which can be either '1' or '*m*' on each end to denote correlated instances of the two artifacts satisfy 1-to-1, 1-to-*m*any, *m*-to-1, or *m*-to-*m* constraint. In Fig. 2, the correlation between `Order` and `Payment` is undirected (the correlation is set up externally) and has the 1-to-1 cardinality constraint (one `Order` instance correlates to exactly one `Payment` instance). The correlations between `Order` and `Purchase` and between `Order` and `Fulfillment` are directed (creation) to denote that an `Order` may create at runtime multiple `Purchase` instances and multiple `Fulfillment` instances.

Naturally, the specified cardinality may contradict with each other in general. Consider the example in Fig. 2, where the cardinality between `Order` and `Payment` is 1-to-1 the cardinality between `Order` and `Purchase` is 1-to-*m*. Suppose we add an edge between `Payment` and `Purchase` with cardinality 1-to-1. Then we will have a problem at runtime when an `Order` instance correlates with two `Purchase` instances and one `Payment` instance: one of the two `Purchase` instances will not be able to correlate with any `Payment` instances. The consistency of cardinality can be checked in linear time [10]. Thus we only consider correlation graphs that have consistent cardinality. The correlation graph shown in Fig. 2 is cardinality consistent.

5 Derived Correlations

In addition to correlations specified in a correlation graph, there may be correlations that are "derived" from existing correlations.

Intuitively, two artifact instances that have no direct correlation (i.e., no edge between these two artifacts in their corresponding correlation graph) can have a "derived correlation" if some rules are satisfied. For our example (Fig. 1 and Fig. 2), a `Fulfillment` instance and a `Purchase` instance may be correlated if they have one item in common (i.e. the *vendor* stores this item bought by a customer through a

correlated `Order` instance, in the *warehouse*). In our choreography language, rules are used to specify derived correlations based on existing (direct or derived) correlations. Thus, whenever a correlation is established, it can be used in rules for specifying other derived correlations.

Before introducing the rules to build derived correlations, we need the concepts of "correlation references", "path expressions", "binary operators", and "quantifiers".

If there is an edge connecting two artifacts in an correlation graph for two artifacts, operator '►' can be used to link these two artifacts to denote all correlated artifacts instances of the current one(s). Consider the example in Fig. 2, expression '`Order`►`Purchase`' denotes all correlated `Purchase` instances of an `Order` instance. Furthermore, suppose o is an instance of `Order`, 'o►`Purchase`' denotes all `Purchase` instances correlated with o. We also denote this expression simply as '`Purchase`$\langle o \rangle$'. The "►" operator can be chained together, crating a *correlation reference*. An example correlation reference is the following:

`Order`►`Fulfillment`►`WarehouseGrouping`,

where `WarehouseGrouping` is an artifact representing a process that collects the items from different locations in the same warehouse.

In general, given a fixed artifact instance, the same correlation reference expression can return different instances due to the fact that some more instances might be created. In Fig. 2, the instance level correlations between `Order` and `Purchase` (`Fulfillment`) are created at runtime. If two artifacts are connected by an undirected edge, their correlated identifier pairs are assumed to exist in the system. In our running example, the instance level correlation between `Order` and `Payment` may be specified by the customer using, e.g., the `Order` ID submitted to the bank.

Path expressions (with the "dot" operator), are used to access hierarchical data [3] of an artifact. In our example (Fig. 1 and Fig. 2), '`Order`.Cart.Inv_ID' is a path expression denoting all inventory IDs of a given `Order` instance. Moreover, the path expressions can contain correlation references. For instance, '`Order`►`Fulfillment`.Item.Name' denotes the item names in all `Fulfillment` instances that are correlated to a given `Order` instance. Similar to correlation references, given o as an instance of `Order`, 'o►`Fulfillment`.Item.Name' denotes the values returned by evaluating path expression '`Order`►`Fulfillment`.Item.Name' but restricted to paths starting from o. An equivalent alternative expression is '`Fulfillment`$\langle o \rangle$.Item.Name'.

To manipulate on the values returned by a path expression, quantifiers (SOME and ALL) and operators such as $\geqslant$, $\neq$, $<$ (comparisons of values) and IN (set membership testing) can be used to forming Boolean conditions. For example, if a `Purchase` instance p and a `Fulfillment` instance f correlate to the same `Order` instance, then 'SOME(`Order`$\langle p \rangle$) IN `Order`$\langle f \rangle$' is true. In this case, since there is exactly one `Order` instance correlated with p and with f, the stronger condition '`Order`$\langle p \rangle$ = `Order`$\langle f \rangle$' is also true. (Note that an operator "$\sqcap$" was used in [10] to express the former condition of having an overlap.)

By using correlation references, path expressions, quantifiers, and comparison operators, we can specify *derived correlations* among artifacts (instances). For example,

in Fig. 2, one valid correlation rule for `Purchase` and `Fulfillment` can be

COR(`Fulfillment`, `Purchase`) :
SOME(`Purchase`.Item.Inventory_ID) IN `Fulfillment`.Item.Inventory_ID

which specifies that a given `Fulfillment` instance has a derived correlation with a given `Purchase` instance if they share at lease one common item. The expression has two parts: (1) 'COR(`Fulfillment`, `Purchase`)' declares a derived correlation between `Purchase` and `Fulfillment`, and (2) the remaining (i.e., the expression after colon) defines a Boolean condition for this correlation. Two artifact instances are correlated if and only if the corresponding Boolean condition evaluates to true on the two instances.

Once a derived correlation is defined, the correlation references can be used upon the corresponding artifacts. Continuing the example discussed in the above, the expression '`Fulfillment`►`Purchase`' can be used even though there is no edge between them in the correlation graph. However, different from edge-specified correlations, derived correlations do not have associated cardinality constraints.

6 Messages

With correlations defined, messages can be sent between two correlated artifact instances. This section describes specifications of message types and instances.

A graphical representation of all messages types are shown as a *message diagram*, where each edge represents a *message (type)* with the edge direction indicating the message flow. A message diagram for our running example is shown in Fig. 3.

Example 6.1 in the following depicts a complete scenario of all messages sending and receiving based on Fig. 3.

Example 6.1 An `Order` artifact instance is created upon receiving an order request (OR) message instance from a requesting customer. The created `Order` instance will then send an invoice (IV) message instance back to the customer. Upon receiving the invoice, the customer is able to send a payment request (PR) message instance to the bank that will make a payment through sending a check (CH) to the `Order` instance. Once the payment is made, the online store can create several (correlated) `Purchase` and `Fulfillment` artifact instances by sending create purchase (CP) and create fulfillment (CF) messages. A purchase complete (PC) is sent from a `Purchase` instance to each correlated `Fulfillment` after a vendor completes the purchase. When all purchases complete, the `Order` instance will request each `Fulfillment` instance to ship the items ordered by the customer by sending request shipping (RS) messages. When items are sent to delivery, shipping complete (SC) messages are sent from warehouses to the online store, which will then close the case by informing the customer through an order complete (OC) messages. ∎

We use an artifact with name '*ext*' (whose structure only contains attribute 'ID') to denote the external environment (as the sender or receiver). Furthermore, the artifact instance of '*ext*' also has ID '*ext*'.

Message names	Abbr.	Sender	Receiver	Data structure	ic	min	max
Order Request	OR	`Ext`	`Order`	OR[ID, Customer[..], Cart*[..], ..]	+	1	1
Invoice	IV	`Order`	`Ext`	IV[ID, Amount, BankInfo[..], ..]	−	1	1
Payment Request	PR	`Ext`	`Payment`	PR[ID, Amount, Customer[..], ..]	+	1	1
Check	CH	`Payment`	`Order`	CH[ID, OrderID, Amount, ..]	−	1	1
Create Purchase	CP	`Order`	`Purchase`	CP[ID, Cart*[Item[Price, ..], ..], ..]	+	1	∞
Create Fulfillment	CF	`Order`	`Fulfillment`	CF[ID, Cart*[Item[..], ..], ..]	+	1	∞
Purchase Complete	PC	`Purchase`	`Fulfillment`	PC[ID, PurchaseID, ..]	−	0	∞
Request Shipping	RS	`Order`	`Fulfillment`	RS[ID, OrderID, ..]	−	0	∞
Shipping Complete	SC	`Fulfillment`	`Order`	SC[ID, FulfillmentID, ..]	−	1	∞
Order Complete	OC	`Order`	`Ext`	OC[ID, OrderID, Date, ..]	−	1	1

Fig. 4. A comlete set of messages

A message contains a message name, a sender artifact name, a receiver artifact name, a complex data structure (which resembles an artifact structure) representing the content of the message, a flag '*ic*' (abbreviation for '*is-creation*') ranging over $\{+, -\}$ to denote if the message can create new artifact instances ('+') or not ('−'), and and minimum and maximum number of message instances that a sender artifact instance can send.

Fig. 4 summarizes all messages for the running example, where the data structure is described using brackets ([ ]) and stars ($*$). For example, OR (order request) message contains child attributes 'ID', 'Customer', and 'Cart', where 'Cart' is a set of tuples; while 'ID' and 'Customer' are non-set attributes. According to Fig. 4, '$CP_{1:\infty}$(`Order`, +`Purchase`) [ID, Cart$*$[Item[Price, ..], ..], ..]' defines a message type from `Order` to `Purchase`. '$1 : \infty$' specifies the minimum (i.e., 1) and maximum numbers (i.e., ∞ or unlimited) of CP message instances that can be sent by an `Order` instance. The '+' symbol indicates that a new receiving instance will be created from each arriving message. The attributes inside '[...]' denote message contents. Similarly, '$PR_{1:1}$(ext, +`Payment`) [ID, Amount, Customer[..], ..]' is a message type whose messages are from the external environment.

The sender and receiver of a message cannot both be '*ext*'; when neither is '*ext*', the two artifacts should have correlation between them (either an edge in the correlation graph, or a derived correlation).

Analogous to artifact instances, a *message instance* is an assignment of values to the corresponding fields in a message type.

7 Choreography Constraints

In this section we introduce "choreography constraints", which specify temporal conditions on message occurrences and may also contain conditions on data in related artifact instances and the messages.

As stated in Example 6.1, the order of messages cannot be arbitrary. Therefore, we need temporal constraints to restrict the behavior of message sending/receiving. In addition to temporal restrictions, conditions on data are also essential. For example, it is feasible for a `Fulfillment` instance to ship the items without receiving the PC

(purchase complete) message from its correlated `Purchase` instance if every item has price less than $100. To specify such conditions involving data, an augmentation of the temporal constraints is needed.

Roughly, we use (non-temporal) "message formulas", which can examine message names and contents as well as the contents of sending/receiving artifact instances. Each constraint then uses a temporal operator to connect two message formulas. Individual LTL operators are not expressive enough while general LTL formulas would make the language rather complicated. We thus use binary operators from DecSerFlow [11] to connect two message formulas.

In order to give an overall view of choreography constraints, we provide Examples 7.1 and 7.2 below that include two complete choreography constraints.

Example 7.1 Consider the following restriction on message sequences: For each OR (order request) sent to a (new) `Order` instance, there is a corresponding CP (create purchase) message in the future sent by the `Order` instance to each correlated `Purchase` instances, and vice versa. The choreography constraint defining the restriction is

$$\forall x \in \texttt{Order}\ \forall y \in \texttt{Purchase}\langle x\rangle\ \textsc{Msg}(\text{OR}, z_{\text{OR}}, \text{ext}, x) \dashv[\textsf{scc}]\mapsto \textsc{Msg}[z_{\text{OR}}](\text{CP}, z_{\text{CP}}, x, y)$$

In the above expression, $\textsc{Msg}(\text{OR}, z_{\text{OR}}, \text{ext}, x)$ and $\textsc{Msg}[z_{\text{OR}}](\text{CP}, z_{\text{CP}}, x, y)$ represent message instances of OR and CP respectively, where z_{OR} and z_{CP} are variables holding the corresponding message instances (IDs). For $\textsc{Msg}(\text{OR}, z_{\text{OR}}, \text{ext}, x)$, ext is the sender and x is the receiver (a variable holding an instance of `Order`) of message z_{OR}; while for $\textsc{Msg}[z_{\text{OR}}](\text{CP}, z_{\text{CP}}, x, y)$, $\textsc{Msg}[z_{\text{OR}}]$ denotes that message instance z_{CP} is a *response* to message instance z_{OR}. The operator '$\dashv[\textsf{scc}]\mapsto$' is "normal succession" that means: if the condition on the left-hand side of $\dashv[\textsf{scc}]\mapsto$ is true, then in the future, the condition on the right-hand side of $\dashv[\textsf{scc}]\mapsto$ is true, and vice versa. ∎

Example 7.2 The following shows a constraint that defines a sequential restriction between messages CP (create purchase) and PC (purchase complete):

$$\begin{aligned}&\forall y \in \texttt{Purchase}\ \forall w \in \texttt{Fulfillment}\langle y\rangle\ \forall x \in \texttt{Order}\langle y\rangle\\ &\quad \textsc{Msg}(\text{CP}, z_{\text{CP}}, x, y) \wedge \textsc{Some}(z_{\text{CP}}.\text{cart.item.price}) > 100\ \dashv[\textsf{rsp}]\mapsto\ \textsc{Msg}(\text{PC}, z_{\text{PC}}, y, w)\end{aligned}$$

The above constraint states the following: for each `Purchase` instance y, each correlated `Fulfillment` instance w of y, and each correlated `Order` instance x of y, whenever x sends a CP (create purchase) message to y that contains at least one item (denoted by quantifier 'SOME') that has price greater than 100, then in the future (denoted by operator '$\dashv[\textsf{rsp}]\mapsto$'), y will send a PC (purchase complete) message to w. (In other words, if all items are priced $\leqslant 100$, this constraint is automatically satisfied.) ∎

Examples 7.1 and 7.2 illustrate that a choreography constraint built with a "temporal operator" connecting two "message formulas" to define a causal relationship. In the remainder of this section, we briefly explain these two main concepts.

Relationships	Operators	Semantics
exist	a –[ex]– b	If a is true, then b must be true in the future or past
co-exist	a –[co-ex]– b	Both a –[ex]– b and b –[ex]– a
normal response	a –[rsp]→ b	If a is true, then b must be true in the future
normal precedence	a –[prc]→ b	If b is true, then a must be true in the past
normal succession	a –[scc]→ b	Both a –[rsp]→ b and b –[prc]→ a
alternative response	a –[al-rsp]→ b	If a is true, then b must be true in the future; and before this b is true, no other a can be true (i.e., a and b should be alternative)
alternative precedence	a –[al-prc]→ b	If b is true, then a must be true in the past; and before this a is true, no other b can be true (i.e., b and a should be alternative)
alternative succession	a –[al-scc]→ b	Both a –[al-rsp]→ b and b –[al-prc]→ a
immediate response	a –[im-rsp]→ b	If a is true, then b must be true immediately after
immediate precedence	a –[im-prc]→ b	If b is true, then a must be true immediately before
immediate succession	a –[im-scc]→ b	Both a –[im-rsp]→ b and b –[im-prc]→ a

Fig. 5. Temporal operators

Temporal operators

Temporal operators in our language are the binary operators from DecSerFlow [11]. These operators define the following 11 binary relationships: exist, co-exist, normal response, normal precedence, normal succession, alternative response, alternative precedence, alternative succession, immediate response, immediate precedence, and immediate succession. The operators and semantics are summarized in Fig. 5.

Message formulas

A message formula is composed of two parts: a message predicate and a conjunction of data conditions.

A message predicate has the name 'MSG' with four parameters: a message name, a message instance (ID), a sender artifact instance (ID), and a receiver artifact instance (ID). Example 7.2 shows two occurrences of the message predicate: MSG(CP, z_{CP}, x, y) and MSG(PC, z_{PC}, y, w), where CP and PC are message names, z_{CP} and z_{PC} are message instances, x and y are senders, and y and x are receivers. MSG(CP, z_{CP}, x, y) denotes that an `Order` instance x sends a CP message instance z_{CP} to its correlated `Purchase` instance y sometime during the execution of the collaborative business process.

There is a variant of message predicates, which has been shown in Example 7.1. MSG[z_{OR}](CP, z_{CP}, x, y) augments the notation of a message predicate by adding a responding correlation MSG[z_{OR}].

Data conditions have similar forms to correlation rules. In Example 7.2, the rule includes the condition "SOME(z_{CP}.cart.item.price)>100" to mean that the message instance z_{CP} should have at least one item with price greater than 100.

In conclusion, choreography constraints are composed of message formulas and temporal operators. Note that all variables in a choreography constraint are always universally quantified. Also, choreography constraints can be expressed in first-order LTL.

8 Semantics

Our declarative choreography language is composed of five components that were introduced in Sections 3 to 7. In this section, we explains the semantics for the choreography language.

The semantics of the language is based on sequences of *system snapshots*. A system snapshot is "captured" whenever a message instance is sent within the collaborative business process, and the snapshot contains the information of all active artifact instances as well as the only *current* message instance. We assume that messages cannot be sent simultaneously.

For example, suppose the current system contains an `Order` instance o, a `Payment` instance m, two `Purchase` instances p_1, p_2, and three `Fulfillment` instances f_1, f_2, f_3. Among them, o is correlated with every instance other than itself, p_1 is correlated with f_1 and f_2, and p_2 is correlated with f_1 and f_3 (through derived correlation rules). Suppose a PC message instance m_{PC} is sent from p_2 to f_1; then the current system snapshot contains all artifact instances together with their correlations and the message instance m_{PC}. We call this snapshot Σ_1.

Now consider the following choreography constraint:

$$\forall y \in \texttt{Fulfillment}\ \forall w \in \texttt{Purchase}\langle y\rangle\ \forall x \in \texttt{Order}\langle y\rangle$$
$$\text{MSG}(\text{PC}, z_{\text{PC}}, w, y) \ \dashv\!\text{rsp}\!\mapsto\ \text{MSG}(\text{RS}, z_{\text{RS}}, x, y) \wedge y.\text{status} = \text{“Not-Yet-Shipped”}$$

which specifies that for each `Fulfillment` instance y, if each `Purchase` instance w correlated to y sends a PC (purchase complete) message to y, then in the future, y's correlated `Order` instance w should send a RS (request shipping) message to y and y's attribute "status" should have the value "Not-Yet-Shipped".

Then, if snapshot Σ_1 occurs in the system, a snapshot Σ_2 must occur some time in the future to satisfy the above constraint where Σ_2 contains the same artifact instances and correlations as the ones in Σ_1, f_1's "status" attribute has the value "Not-Yet-Shipped", and the current message sent in Σ_2 is a message instance of RS from o to f_1.

9 Conclusions

This paper uses an example to illustrate the declarative choreography language proposed in [10]. The language can express correlations and choreographies for artifact-centric BPs in both type and instance levels. It also incorporate data contents and cardinality on participant instances into choreography constraints. Reference [10] also includes discussion on realizability, i.e., how to implement a choreography, and on further research issues.

References

1. Web Services Business Process Execution Language (BPEL), Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`, 2007.
2. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *Proc. of the 12th Int. Conf on World Wide Web, WWW*, pages 403–410, 2003.

3. R. Cattell and D. Barry. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.
4. G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *Proc. of the 5th Int. Conf. on Web Services, ICWS*, 2007.
5. J. Hanson, P. Nandi, and S. Kumaran. Conversation support for business process integration. In *Enterprise Distributed Object Computing Conference, 2002. EDOC '02. Proceedings. Sixth International*, pages 65 – 74, 2002.
6. R. Hull and J. Su. Tools for composite web services: a short overview. *SIGMOD Record*, 34(2):86–95, 2005.
7. R. Hull, J. Su, and R. Vaculín. Data management perspectives on business process management: tutorial overview. In *SIGMOD Conference*, pages 943–948, 2013.
8. A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.
9. C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10):46–52, 2003.
10. Y. Sun, W. Xu, and J. Su. Declarative choreographies for artifacts. In *ICSOC*, pages 420–434, 2012.
11. W. M. P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *Proc. of the 3rd Int. Workshop on Web Services and Formal Methods, WS-FM*, pages 1–23, 2006.
12. J. M. Zaha, A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In *Proc. of the 14th Int. Conf. on Cooperative Information Systems, CoopIS*, pages 145–162, 2006.