Declarative Choreographies for Artifacts*

Yutian Sun[1], Wei Xu[2,**], and Jianwen Su[1]

[1] Department of Computer Science, UC Santa Barbara, USA
[2] School of Computer Science, Fudan University, China

Abstract. A choreography models a collaboration among multiple participants. Existing choreography specification languages focus mostly on message sequences and are weak in modeling data shared by participants and used in sequence constraints. They also assume a fixed number of participants and make no distinction between participant type and participant instances. Artifact-centric business process models give equal considerations on modeling both data and control flow of activities. These models provide a solid foundation for choreography specification. This paper makes two contributions. First, we develop a choreography language with four new features: (1) Each participant type is an artifact schema with (a part of) its information model visible to choreography specification. (2) Participant instance level correlations are supported and cardinality constraints on such correlations can be explicitly defined. (3) Messages have data models, both message data and artifact data can be used in specifying choreography constraints. (4) The language is declarative based on a mixture of first order logic and a set of binary operators from DecSerFlow. Second, we develop a realization mechanism and show that a subclass of the choreography specified in our language can always be realized. The mechanism consists of a coordinator running with each artifact instance and a message protocol among participants.

1 Introduction

Collaborative business processes (CBPs) are a necessity for businesses to stay competitive [8,11]. A recent study reports that an overwhelming majority of eCommerce volume is associated with B2B collaboration [13]. CBPs involve multiple participants, and multiple resources spread over multiple administrative domains. Typically CBPs are complex in terms of process logic, relationships among participants and resources, distributed execution, and semantic mismatches between participant data, ontologies, and behaviors. Such complexity is the source of many technical difficulties in design, analysis, realization, execution, and management of CBPs. Tools and support for CBPs continue to be a major challenge in current and future enterprise [13].

CBPs can be divided into two classes. An *orchestrated* CBP uses a designated "mediator" to communicate and coordinate with all participating BPs (business processes). Although this approach is widely used in practice (e.g., for cross-organizational workflows), it loses autonomy of participating BPs and does not scale well. The *choreography* approach specifies global behaviors among participating BPs but otherwise leaves

* Supported in part by NSF grant IIS-0812578 and grants from IBM and Bosch.
** Part of work done while visiting UCSB.

C. Liu et al. (Eds.): ICSOC 2012, LNCS 7636, pp. 420–434, 2012.

the BPs to operate autonomously and communicate in the peer-to-peer fashion. Technical difficulties for this approach include the lack of suitable choreography specification language(s) and mechanisms to coordinate among participating BPs in absence of a central control point. This paper develops a language for CBP choreography specification and addresses the coordination issue.

A choreography models a collaboration among multiple participants. A choreography may be specified as a state machine representing message exchanges between two parties [9] or permissible messages sequences among two or more parties with FIFO queues [2], or as a process algebra expression with sequence, parallel, conditional, and loop constructs [3,4]. It may be specified in individual pieces using patterns [24], as a composition of message interactions [19], or implicitly through participants behaviors [6]. WS-CDL proposes an XML-based package for specifying choreography through a conventional set of control flow constructs over messaging activities. Existing choreography languages focus mostly on specifying message sequences and are weak in modeling data shared by participants and used in choreography constraints. WS-CDL models data through variables that only implicitly associated with participants that produce data. A tightly integrated data model with message sequence constraints would allow a choreography to accurately constrain execution. Also, these languages assume a fixed number of participants and makes no distinction between a participant *type* and a participant *instance*. For example, an *Order* process instance may communicate with ***many*** *Vendor* process instances; this cannot be effectively captured without the type vs. instance distinction. Therefore, a choreography language should be able to model correlations between process instances. Existing languages either do not support such correlations, or lack the ability to reference instance correlations in a choreography.

In recent BPM research, artifact-centric BP models [17] have attracted increasing attention. An artifact-centric BP model includes (*i*) an information model for business data, (*ii*) a specification of lifecycle that defines permitted sequences of activities, and (*iii*) a data model for keeping track of runtime status and dependencies. Artifact-centric BP models provide a technical foundation for choreography specification.

This paper focuses on choreography specification and realization, and makes the following two technical contributions. **First**, we develop a choreography language with four distinct and new features: (1) Each participant type is an artifact schema (BP model) with a selected sub-part of its *information model* visible to choreography specification. (2) Correlations between participant types and *instances* are explicitly specified, along with *cardinality constraints* on correlated instances (e.g., each *Order* instance may correlate with exactly one *Payment* instance and multiple *vendor* instances). In particular, Skolem notations are used to reference correlated participant instances. (3) Messages can include data; both message data and artifact data can be used in specifying choreography constraints. Also, Skolem notations are used to manipulate dependencies among messages and participant instance created by messages. (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 [22]. **Second**, we formulate a distributed algorithm that realizes a subclass of the choreography in our language. This subclass contains choreographies allowed by most existing languages including conversation protocols [21]. Specifically, a choreography is translated to an equivalent finite state machine. Based

on this global state machine, our distributed algorithm coordinates participant instances using a simple protocol. We show that the set of global behaviors of coordinated execution of participant instances by our algorithm coincides with the set of behaviors permitted by the choreography. Furthermore, the total number of messages in each successful run is at most twice the number of messages needed for the collaborative process execution plus the number of participants.

This paper is organized as follows. A motivating example is provided Section 2. The choreography language and the realization results are presented in Sections 3 and 4 (resp.). Related work is discussed in Section 5. Conclusions are in Section 6.

2 Motivating Example

This section illustrates with an example concepts including artifact-centric BPs and motivates the need for specification of correlations among process instances and choreographies with data contents.

Consider an online *store* where items are available at *vendor*s. 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 (1) the items in her cart by warehouses and sends to each warehouse for fulfillment, and (2) the items by vendors and requests each vendor to complete the purchase. The vendors inform warehouses upon completion of the purchases. After the store receives the payment and vendors' completion of purchases, the store asks warehouses to proceed with shipments.

In this example, four types of participants (*store*, *vendor*, *warehouses*, and *bank*) are involved and each type has its own BP. Although *store* and *bank* have only one process instance each for a single shopping session, there may be multiple instances of *vendor* and of *warehouse*. In artifact-centric modeling, an artifact instance encapsulates a running process. For example, *store* initiates an "`Order`" (artifact) instance. Fig. 1 shows a part of the structured data in an `Order` instance. The structure contains attributes ID, (shopping) Cart, etc., where Cart is a relation-typed attribute (denoted by "$*$") that may include 0 or more tuples with four nested attributes: Inv(entory)_ID, (item) Name, Quan(tity), and Price. Similarly, other participant BPs 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*.

Consider specifying a choreography for the CBP in this example, there are two major difficulties. First, existing languages do not support multiple participant instances, and thus the fact that multiple vendor/warehouse instances cannot be easily represented and included in specifying behaviors. Some process algebra based languages allow instantiation of new instances from sub-expressions in a choreography [3,4], but it is not clear how it is related to multiple participant instances. Second, behaviors often depend on data contents. For example, when an order request is received with total amount $\geqslant$10, the order processing should proceed as described in the above; for orders with amount <10, the processing may be optional. Such conditions on data cannot be easily expressed in most languages. WS-CDL may express this through copying messages to variables, but copying adds unnecessary data manipulations.

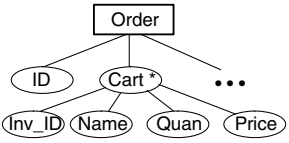


Fig. 1. Data structure

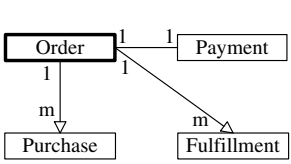


Fig. 2. Correlation diagram

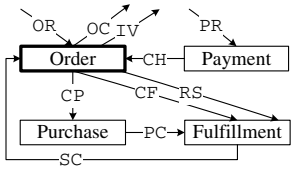


Fig. 3. Message diagram

3 A Declarative Choreography Language

This section introduces a declarative language for defining choreographies. In this language, a choreography assumes participant BPs are modeled as artifacts and consists of correlations between artifacts and instances, messages, and a set of choreography constraints based on first-order linear temporal logic. §3.1 defines "collaborative schemas" containing correlations and messages, and choreography constraints are defined in §3.2.

3.1 Correlations of Artifacts, Messages, and Collaborative Schemas

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

Definition: An *artifact (interface)* is a pair (ν, Att), where ν is a (distinct) name and Att a set of attribute names (or attributes) whose values may be hierarchically structured.

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 instance.

Given an artifact interface $\alpha = (\nu, Att)$, an *artifact instance* I of α is a partial mapping from Att to the corresponding domains such that $I.\text{ID}$ is defined and unique.

We now define an important notion of a "correlation graph". Intuitively, such a graph specified whether instances of two BPs are correlated and whether the correlation is one instance of a BP correlating to one or many instances of the other BP. Similar to WS-CDL, only a pair of correlated instances may exchange messages in our model.

Definition: A *correlation graph* G is a tuple (V, ρ, E, C, λ), where

- V is a nonempty set of artifact interfaces closed under references (through attributes). We may call artifact interfaces in V "nodes" (of the graph),
- $\rho \in V$ is the *primary* artifact interface (the root),
- $E \subseteq V \times V$ is symmetric denoting correlations (undirected edges) among artifact interfaces such that (V, E) is connected and contains no self-loops,
- $C \subseteq E$ is asymmetric denoting creation relationships among artifact interfaces such that (i) for each $v \in V$ there exists at most one pair $(u, v) \in C$ (can only be created once), (ii) (V, C) is acyclic (no cyclic creations), (iii) there is no $v \in V$ such that $(v, \rho) \in C$ (primary instances can only be created by external messages), and
- λ is a partial mapping from $E \times V$ to $\{1, \text{m}\}$ (cardinality of correlations) such that
  - $\lambda((u, v), v') \in \{1, \text{m}\}$ iff v' is an end node of $(u, v) \in E$,
  - for each $(\rho, v) \in E$, $\lambda((\rho, v), \rho) = 1$ (single primary instance),

- for each $(u, v) \in C$, then $\lambda((u, v), u)$ is 1 (no multiple creation), and
- for each $(u, v) \in (E - C)$ where $(v, u) \notin C$, $\lambda((u, v), u) = \lambda((v, u), u)$ (consistency for undirected edges).

Intuitively, a correlation graph models correlations among artifacts and artifact instances. More precisely, if two artifact interfaces are correlated (connected by an edge), it indicates that some instances of these two artifact interfaces are correlated. The mapping λ indicates the type of cardinality of instances (1-to-1, 1-to-*m*any, *m*-to-1, *m*-to-*m*). Fig. 2 shows a correlation graph for the example in Section 2 with 4 nodes and 3 edges (2 directed and 1 undirected) with cardinality constraints. The artifact interface `Order` is primary. The directed edges indicate creation of instances, e.g., an `Order` instance would create multiple `Purchase` instances and multiple `Fulfillment` instances.

If there is an edge (α_1, α_2) in a correlation graph for two artifact interfaces α_1 and α_2, correlations of their instances can be represented by a binary relation *Corr*. *Corr* may change at runtime, but it must always satisfy cardinality constraints in a correlation graph. Given an identifier (ID) y of an artifact instance of α_2, the notation $\alpha_1\langle y\rangle$ denotes the set $\{x \mid \exists y Corr(x, y)\}$ in the current "system state" (details provided in §3.2).

Example 31. Consider the example in Fig. 2, if o is the ID of a `Purchase` instance, the expression "`Order`$\langle o\rangle$" is the ID of the correlated `Order` instance. ∎

The relation *Corr* changes when one instance creates another (a directed edge in a correlation graph). 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 *Corr*. 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.

In addition to correlations specified in a correlation graph, there may be correlations that are "derived" from existing correlations. For example, a `Purchase` instance is correlated with a `Fulfillment` instance if they have a common item in their correlated `Order` instance. Derived correlations will be defined with rules that use "path expressions" and the "intersection" predicate.

Path expressions (with the "dot" operator) are used to access hierarchical data [5]. Technically, given an artifact interface $\alpha = (v, Att)$, a path expression of α is of form "$v.A_1.A_2.\cdots.A_n$", where $A_1 \in Att$ and each A_{i+1} ($i \in [1..(n-1)]$) is an attribute nested in A_i.

Example 32. Continue with the example in Section 2, a path expression could be "`Order`.Cart.Inv_ID", which corresponds to the structure shown in Fig. 1. ∎

If a path expression e returns an identifier of an instance, $e.A$ will return the value of attribute A in the instance. In general a path expression may return a set of values, similar to XPATH expressions.

We use a (binary) *intersection* predicate "$\sqcap$" to check if two input sets have nonempty overlap. We also apply the predicate on individual values treating a value as a singleton set. The predicate is conveniently generalized to more than 2 inputs.

Example 33. If a `Purchase` instance I_P (identifier) and a `Fulfillment` instance I_F correlate to the same `Order` instance, "`Order`$\langle I_P\rangle \sqcap$ `Order`$\langle I_F\rangle$" is true. ∎

An *atomic condition* is an intersection condition applied to path expressions. Here we use interface name to denote the identifier of an arbitrary instance of the interface. A *correlation condition* is a set (conjunction) of atomic conditions.

Definition: Given a correlation graph $G = (V, \rho, E, C, \lambda)$, and two artifact interfaces $\alpha_1, \alpha_2 \in V$ where $(\alpha_1, \alpha_2) \notin E$, a *correlation rule* of α_1 and α_2 is of form "$\text{COR}(\alpha_1, \alpha_2) : c$", where c is a correlation condition.

Given artifact interfaces α_1 and α_2, a correlation rule $\text{COR}(\alpha_1, \alpha_2) : c$, and two instance IDs I_1 of α_1 and I_2 of α_2, I_1 is *correlated* with I_2 if c is true on I_1 and I_2.

Example 34. The rule below defines correlation between `Fulfillment` and `Purchase`:

$$\text{COR}(\texttt{Purchase}, \texttt{Fulfillment}): (\texttt{Order}\langle\texttt{Purchase}\rangle \sqcap \texttt{Order}\langle\texttt{Fulfillment}\rangle) \wedge (\texttt{Purchase.item.inventory\_ID} \sqcap \texttt{Fulfillment.item.inventory\_ID})$$

Note that `Purchase` (`Fulfillment`) in the above represents an arbitrary instance. ∎

Once a rule $\text{COR}(\alpha_1, \alpha_2) : c$ is defined, the notation $\alpha_1\langle\alpha_2\rangle$ (or $\alpha_2\langle\alpha_1\rangle$) can be used in the same way as other correlations. Derived correlations do not have cardinality constraints specified.

A rule $\text{COR}(\alpha_1, \alpha_2) : c$ *depends on* another rule $\text{COR}(\alpha_3, \alpha_4) : c'$, if the condition c contains $\alpha_3\langle\alpha_4\rangle$ or $\alpha_4\langle\alpha_3\rangle$.

We now define the messages among the artifacts, using "ext" to denote the external environment (as the sender or receiver).

Definition: Given a correlation graph $G = (V, \rho, E, C, \lambda)$ and a correlation rule set Γ, a *message type* m wrt G, Γ is a tuple $(M, \nu_s, \nu_r, \pi, \phi)$, where

- M is a name;
- ν_s and ν_r are distinct artifact interfaces denoting the sender and receiver (resp.) such that at most one of them can be "ext", and if both are in V, they must be correlated (via an edge in G or by a correlation rule in Γ);
- π is a set of attributes (holding a payload);
- ϕ is "+" (the sending instance creates an instance of the receiving artifact upon arrival of each message instance) or "−" (no creation).

Fig. 3 shows a message diagram, each edge represents a message type with the edge direction indicates the message flow.

Example 35. Continue with the example in Section 2, "CP(`Order`, +`Purchase`) [*OrderID*, *Amount*,...]" defines a message type from `Order` to `Purchase`. (We use a slightly modified syntax for readability.) The "+" symbol indicates that a new receiving instance will be created by each arriving message. The attributes inside "[...]" denote message contents. "PR(ext, +`Payment`) [*OrderID*, *Amount*...]" is a message type whose messages are from the external environment. ∎

A *message instance* of a message type $(M, \nu_s, \nu_r, \pi, \phi)$ is of form $M(id_m, id_s, id_r, \mu)$, where id_m is a unique message ID, μ is a mapping from π to the corresponding domains, and id_s and id_r are the IDs of instances of α_s and α_r (resp.) such that if ν_s (ν_r) is "ext" id_s (resp. id_r) is also "ext".

Definition: A *collaboration schema* is a tuple (G, Γ, Msg), where G is a correlation graph; Γ is a set of correlation rule of G whose rule dependencies are acyclic, and Msg is a set of message types wrt G, Γ.

Roughly, a collaboration schema defines the correlations among artifact interfaces (participant types) and among instances (participants), and the message types.

3.2 Choreography Constraints

Roughly, a specification consists of "choreography constraints", each stating a temporal property on message occurrences and may also contain conditions on data in related artifact instances and the messages.

In the technical discussions, we make the following assumptions concerning identifiers. For each artifact interface or message type (name), there is a countably infinite set of artifact or message (resp.) instance IDs; furthermore, these ID sets are pairwise disjoint. Let $\mathbf{ID}_{\mathrm{A}}$ ($\mathbf{ID}_{\mathrm{M}}$) be the union of all artifact (resp., message) instance ID sets. We further assume the existence of three countably infinite sets: an irreflexive *artifact correlation* set $\textbf{Corr} \subseteq \mathbf{ID}_{\mathrm{A}}^2$, a *message-artifact dependency* set $\mathbf{MA} \subseteq \mathbf{ID}_{\mathrm{M}} \times \mathbf{ID}_{\mathrm{A}}$, and an irreflexive *message-message dependency* set $\mathbf{MM} \subseteq \mathbf{ID}_{\mathrm{M}}^2$.

The correlation set **Corr** captures correlations among artifact instances. The message-artifact dependency set **MA** holds dependencies of an arriving message ID that causes creation of an artifact ID. The message dependency set **MM** represents the relationships between messages, e.g., one message may depend another based on contents, or simply request-response. For example, an invoice message may respond to an order request.

For executable languages, it is necessary to define how the sets **Corr**, **MA**, **MM** are created and maintained at runtime. However, a choreography language provides only a specification, i.e., conditions that must be satisfied by every execution. Due to this reason we do not specify how these sets are created and maintained, instead, we assume that they are predetermined and fixed. Conditions will be provided to ensure consistency of the sets and runtime correlations/dependencies among artifact/message instances.

We now define "system states" that represent snapshots at time instants. Note that we require all artifact instances in a system state must be correlated directly or indirectly. This restriction allows us to focus on a single collaboration instance.

Definition: For a collaboration schema $C = (G, \Gamma, Msg)$, a *system (s-)state* of C is a triple $(S, \overline{\mathrm{M}}, m)$, where S is a set of artifact instances for G, $\overline{\mathrm{M}}$ a finite set of message IDs, m a message instance of a message type M in Msg such that (1) the ID of m is in $\overline{\mathrm{M}}$, (2) if m's sender is not "ext", the sender instance is in S, (3) if M is artifact creation, m's receiver ID is not in S and the message and receiver IDs are in **MA**, (4) if M is not creation, either m's receiver is "ext" or has ID in S, (5) if neither sender or receiver is "ext", they are correlated according to **Corr**, (6) for each correlation rule in Γ and each pair of artifact instance IDs in S, if the IDs satisfy the rule condition, they are correlated in **Corr**, letting S' be the set of all artifact IDs in S or m, (7) the graph $(S', \textbf{Corr})$ is connected and satisfies all cardinality conditions in G, and (8) the graphs $(\overline{\mathrm{M}} \cup S', \mathbf{MA})$ and $(\overline{\mathrm{M}}, \mathbf{MM})$ encode functions (each node has $\leqslant 1$ outgoing edge).

An s-state is *initial* if S is empty, $\overline{\mathrm{M}}$ a singleton set, m is from "ext" to the primary artifact interface.

An s-state is a snapshot of artifact instances, past message IDs, the current message sent. Conditions (2)(4) demand that the sender and receiver are existing artifact instances if not external for non-creation message types. Conditions (3)(5)(6) concern correlations

and dependencies. The connectivity and cardinality condition (7) requires that each pair of artifact instance IDs is correlated via one or more correlations and all cardinality constraints in the correlation graph hold. Finally condition (8) ensures that each message creates at most one artifact and/or depends on at most one message.

Let $(S, \overline{\mathrm{M}}, m)$ be an s-state and S' all artifact IDs in S or m. If an artifact instance I in S' is correlated to instances $I_1, ..., I_n$ in S' of artifact interface α according to **Corr**, the notation $\alpha\langle I\rangle$ denotes the set $\{I_1, ..., I_n\}$ in the s-state. For each message ID μ in $\overline{\mathrm{M}}$ if $(\mu, I_1) \in \mathbf{MA}$ where I_1 is the ID of an instance of artifact interface α and in S', the notation $\alpha[\mu]$ denotes I_1 in the s-state. And if $(\mu, \mu') \in \mathbf{MM}$ for $\mu, \mu' \in \overline{\mathrm{M}}$ and the message type of μ is M, $M[\mu']$ has the value μ (a reply to μ') in the s-state.

For a collaboration schema $C = (G, \Gamma, Msg)$, a *system (s-)behavior* of C is a finite sequence $\sigma_1\sigma_2\cdots\sigma_n$ of s-states of C such that σ_1 is initial, messages in σ_i's have distinct IDs, and for each $i \in [1..(n-1)]$, the following conditions all hold for $\sigma_i=(S_i, \overline{\mathrm{M}}_i, m_i)$ and $\sigma_{i+1}=(S_{i+1}, \overline{\mathrm{M}}_{i+1}, m_{i+1})$: every artifact ID in S_i is in S_{i+1} and every artifact ID in S_{i+1} is either in S_i or the receiving ID of m_i, and $\overline{\mathrm{M}}_{i+1} = \overline{\mathrm{M}}_i \cup \{m_{i+1}\}$ (m_{i+1} denotes its ID).

Intuitively, an s-state advances by consuming the current message (instance) and producing the next message. If the receiving ID does not correspond to an artifact instance, a new instance is created. The changes of data contents of artifact instances are the responsibility of participant processes and thus not captured in s-state transitions. Also, message-message dependency is not required, creating such dependencies is also done by individual participant BPs.

We now focus on "choreography constraints". Roughly, we apply (non-temporal) "message formulas" to s-states which examines message type 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, therefore we use binary operators from DecSerFlow [22].

Let C be a collaboration schema. Each message type (name) is a ternary *message predicate*. For each s-state σ, and a message type M, $M(\mu, a, b)$ is *satisfied* in σ if μ, a, b are the IDs of the message instance, its sending and receiving artifact (resp.) in σ. Note that Skolem notations can be used to indicate dependencies. For example, $M(M[\mu], a, b)$ is true in s-state σ the response message of type M to the message μ is sent from a to b. For notational convenience, we will abbreviate "$M(M[\mu], a, b)$" as "$M[\mu](a, b)$".

Starting from artifact/message IDs, path expressions can be used to access attribute values, hierarchically organized values, and other artifacts whose IDs are stored as attribute values. Built-in relational comparisons can be used to test results of path expressions. Given an s-state, satisfaction of *data conditions* with path expressions is defined in the standard manner. If an attribute value is not defined, a guarded approach such as in [10] can be used. Data conditions are used in conjunction with message predicates.

Example 36. The formula "$\mathrm{CP}(\mu, I_\mathrm{O}, I_\mathrm{P}) \wedge \mu.\mathrm{cart.price} > 100$" checks if the message μ from `Order` instance I_O to `Purchase` instance I_P has an item with price > 100. ∎

Our logic language includes variables ranging over $\mathbf{ID}_A$ and $\mathbf{ID}_M$ and message predicates, data conditions with variables. Given a collaboration schema C, a *message formula of* C is of form $\Phi \wedge (\bigwedge_{i=1}^{n} \varphi_i)$, where Φ is a message predicate and for each φ_i is a data condition with path expressions and Skolem notions built from variables in Φ.

Let C be a collaboration schema. a *choreography constraint of* C is an expression of form $\Psi_1 \Theta \Psi_2$ where Ψ_i's are message formulas and Θ is one of the following temporal operators from DecSerFlow [22]: exist, co-exist, normal response, normal precedence, normal succession, alternative response, alternative precedence, alternative succession, immediate response, immediate precedence, and immediate succession.

Variables in a choreography constraint are universally quantified and range restricted to the types and the current s-state. We use examples to illustrate the constraints in the remainder of the section. Operators in DecSerFlow can be translated into LTL [22]. The semantics for choreography constraints are rather technical and omitted here. Roughly, choreography constraints can be expressed in first order logic with LTL.

Example 37. Consider the restriction on message sequences for the example in Section 2: For each order-request (OR), with amount greater than 10, sent to a (new) `Order` instance, there is a corresponding create-purchase (CP) message in the future sent by the order instance to all correlated `Purchase` instances, and vice versa. The choreography constraint defining the restriction is

$$\forall x{\in}\text{Order OR}(\mu, \text{ext}, x) \wedge \mu.\text{amount}{>}10 \ \ (succ){\rightarrow} \ \ \text{CP}[\mu](x, \texttt{Purchase}[\mu])$$

Here CP[μ] and `Purchase`[μ] denote the CP message instance and the `Purchase` artifact instance caused by μ. The operator "p $(succ)\rightarrow$ q" is normal succession that means: each p is followed by a q (possibly not immediate) and each q is preceded by a p (possibly not immediate). ∎

Example 38. Consider the restriction that for each order, if there is an item with price is >100, then no ready-to-ship (RS) message is sent until all purchase-complete (PC) messages have been sent. This can also be expressed using normal succession:

$$\forall x{\in}\text{Fulfillment}\forall y{\in}\text{Purchase}\langle x\rangle \text{PC}(\mu, y, x) \wedge y.\text{cart.price}{>}100 (succ){\rightarrow} \text{RS}[\mu](\texttt{Order}\langle x\rangle, x)$$

Here `Order`$\langle x\rangle$ denotes the correlated `Order` artifact instance of x. Similarly, RS[μ] denotes an RS message depending on μ. ∎

Definition: A *choreography (specification)* is a pair (C, κ) where C is a collaboration schema and κ a set of choreography constraints over C.

"Satisfaction" of a choreography $\mathcal{S} = (C, \kappa)$ by an s-behavior of C is defined based on the above discussions. The *semantics* of of $\mathcal{S}$ is the set of all s-behaviors of C that satisfy all constraints in κ.

4 Realizability

In this section, we show that a subclass of choreographies defined in Section 3.2 can be realized. This is accomplished in two stages, we first translate a choreography into a "guarded conversation protocol" that is a conversation protocol of [1] extended with data contents and conditions. We then present a distributed algorithm that runs along with execution of each artifact, and show that an s-behavior is a possible execution with the algorithm iff it satisfies the choreography.

4.1 Guarded Conversation Protocols

A choreography $\mathcal{S} = (C, \kappa)$ is *one-to-one* (or *1-1*) if the correlation graph in C only has 1-1 correlations. The class of 1-1 choreographies roughly corresponds to choreographies definable in most existing languages, with a possible exception of BPEL4Chor [6,12]. We focus on 1-1 choreographies in this section.

Definition: A *guarded (conversation) protocol* is a tuple (T, s, F, M, C, δ), where (i) T is a finite set of states, $s \in T$ is the initial state, $F \subseteq T$ is a set of final states; (ii) M is a finite set of messages type names, (iii) C is a set of data conditions, and (iv) $\delta \subseteq T \times M \times C \times T$ is a set of transitions.

A guarded protocol extends conversation protocol of [7] with data conditions on messages (and associated artifacts). The semantics of a guarded protocol is standard except that the data condition must be satisfied when making a transition.

Example 41. Fig. 4 shows an example guarded protocol with four states: t_2 is initial and t_2, t_4 are the final states. Two message types are involved, X and Y. C_X and C_Y are data conditions. The transition from t_2 to t_3 can be made if the condition C_X is true and message X is sent. An edge labeled with "else" stands for a collection of transitions other than the specified one(s) leaving the same state. An edge labeled with "*" represents all possible transitions leaving the state. ∎

Given an s-behavior B (of a correlation schema), and a guarded protocol τ, the notion of τ *accepting* B is defined in the standard way.

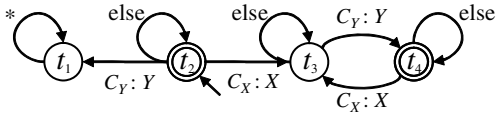


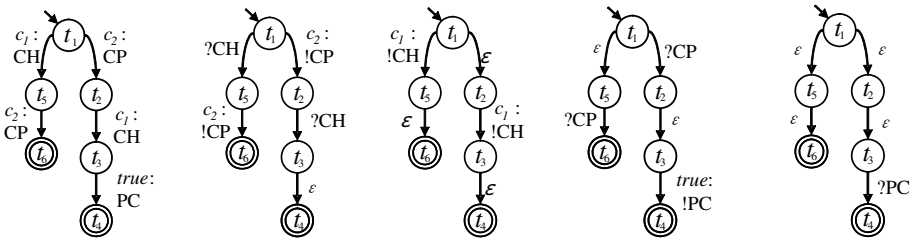
Fig. 4. A guarded conversation protocol example

Theorem 42. Let $\mathcal{S}$ be a 1-1 choreography. One can effectively construct a guarded conversation protocol τ such that each s-behavior B satisfies $\mathcal{S}$ iff it is accepted by τ.

Since temporal operators in choreography constraints are operators in DecSerFlow that is contained in LTL [22], one can use a general technique to obtain Büchi automaton [23]. Guarded protocols can then be constructed. However, we use a simpler approach: translating each choreography constraint to a guarded protocol and then construct a product state machine for all constraints. Fig. 4 shows a guarded protocol for constraint "$X \wedge C_X(succ) \rightarrow Y \wedge C_X$", where X, Y are message predicates and C_X, C_Y data conditions.

Example 43. Fig. 5(a) shows a part of the guarded protocol translated from Example 38, where c_1 is "Payment.balance > CH.amount" and c_2 is "CP.items ≠ null". Since each participant can have at most one instance (1-1 choreography) type level notation is used here. The initial state is t_1, the final states are t_4 and t_6 (in the original guarded protocol, they are not final states but we make them final to show a complete example). Only two sequences of messages can be accepted in this example: either (1) CP CH PC, or (2) CH CP. Note that this is not realizable in [7]. ∎

The only-if direction of Theorem 42 fails if 1-1 condition on choreography is removed. This is because different instances of the same interface may progress in different paces and a guarded protocol cannot capture such situations.



(a) Model (b) `Order` (c) `Payment` (d) `Purchase` (e) `Fulfillment`

Fig. 5. A translated guarded protocol and project to artifacts

4.2 Guarded Peers

Guarded automaton was introduced in [7] to represent a state machine for a participant. We modify the notion to allow message predicates and data conditions. The following defines a projection of a guarded protocol to a participant.

Definition: Given a guarded conversation protocol $\tau = (T, s, F, M, C, \delta)$ and an artifact type α, a *guarded peer for* α wrt τ is a tuple $(T, s, F, M', C', \delta_s, \delta_r, \delta_\varepsilon)$, where (1) $M' \subseteq M$ such that each message in M' has α as a sender or receiver, (2) $C' \in C$ contains a condition c if there exists $t, t' \in T$ and a message m in M', where m is sent by α and $(t, c, m, t') \in \delta$, (3) $\delta_s \subseteq T \times C' \times M' \times T$ (sending transitions) contains elements (t, c, m, t') if $(t, c, m, t') \in \delta$ and m is sent by α, (4) $\delta_r \subseteq T \times M' \times T$ (receiving transitions) contains elements (t, m, t') if there exists $c \in C$, $(t, c, m, t') \in \delta$ and the receiver of m is α, and (5) $\delta_\varepsilon \subseteq T \times \{\varepsilon\} \times T$ (empty transitions) contains elements (t, ε, t') if there exists $c \in C$, $m \in M$, $(t, c, m, t') \in \delta$ and α is neither the receiver or sender of m.

Example 44. Fig. 5(b)–(e) show four guarded peers (`Order`, `Payment`, `Purchase`, and `Fulfillment`) projected from Fig. 5(a). The "?" mark denotes receiving a message, the "!" mark denotes sending a message. ∎

An artifact (instance) sends or receives messages according to its guarded peer, i.e., each guarded peer is autonomous. If all guarded peers start from their initial states, make their transitions autonomously, the composition terminates when every guarded peer reaches a final state. Our composition model is basically the same as [7], except that FIFO queues are not used.

Example 45. Consider the sequence of messages "CH CP" that is accepted by the guarded protocol in Fig. 5(a) (Example 43). The projected peers are shown in Fig. 5(b)–(e). `Payment` can send a CH to `Order`. Then `Payment` follows an empty transition into its final state t_6 and `Payment` is now in t_5. Later on, `Order` sends a CP to `Purchase` and both of them can reach their final states (t_6). While for `Fulfillment`, it sends or receives nothing and follows two empty transitions to t_6. ∎

Naturally, given a guarded protocol τ, if a sequence of transitions is accepted by τ, the sequence is also accepted by all guarded peers of τ. In general, the other direction may not necessarily hold [2].

Example 46. Continue with Examples 43 and 44. Suppose `Payment` sends a CH through edge (t_1, t_5) and ends at final state t_6. Then `Order` sends a CP, receives the CH sent by `Payment`, and reaches final state t_4. Correspondingly, `Purchase` receives the CP from `Order` and reaches final state t_4 by sending `Fulfillment` a PC. Finally, `Fulfillment` receives the PC and ends at t_4 as well.

Clearly, the above sequence of messages "CH CP PC" allows all guarded peers to reach their final states but cannot be accepted by the original guarded protocol. ∎

The *realizability problem* is to ensure that the collective transitions for all guarded peers are equivalent to transitions for the original guarded protocol. While this problem has not been investigated, a closely related problem of "realizability checking problem" [2] which tests if a conversation protocol can be restored from the product of its projected peers has been studied extensively (see [21]).

4.3 A Realization Mechanism

Instead of checking if a guarded protocol is the product of its guarded peers, we take a different approach. We develop a protocol (algorithm) that in addition to the original messages, it also adds a small number of "synchronization" messages to aid participants (peers) in their autonomous execution. We show that the synchronized execution generates equivalent behaviors as the original guarded protocol and that in every successful execution, the total number of synchronization messages is bounded by the sum of the number of messages in the guarded protocol and the number of guarded peers.

A naïve protocol simply broadcasts every message to all. However, this approach requires as many as $N^* \times (k-1)$ messages during the process, where N^* is the number of message instances (needed for the collaboration), and k is the number of peers.

To reduce synchronization messages, an improvement is developed that employs a "token passing" method: only the participant who owns a "token" can make a transition. Once a transition is conducted (or equivalently, a message is sent), the "token" will be passed to the next sender and this process repeats.

Given a guarded protocol $\tau = (T, s, F, M, C, \delta)$, we augment τ with a new message type named *sync* without any data attributes. We also introduce two functions, *Flag* and *State*. The function *Flag* maps message (including sync) instances to the set $\{\text{SND}, \text{RCV}, \text{FIN}\}$ such that if μ is an instance of sync, $\mathit{Flag}(\mu) \in \{\text{SND}, \text{FIN}\}$. Intuitively, SND is the token, RCV means the message that is regular, FIN instructs the receiver to terminate. The function *State* maps each message instance to T to indicate the current (global) state. Each message is sent along with its *Flag* and *State* values.

To implement the framework, a coordinator is used for each peer (instance) to help on transition decisions. Once a coordinator receives the token carried by a message, it makes a transition for its peer by sending a message with an appropriate *Flag*, and possibly passes the token to the next sender if different (via a flagged sync message).

As mentioned at the beginning of this section, once (the coordinator of) a sender sends a message with the flag RCV, it passes the token to (the coordinator of) the next sender through a message with the flag SND. In order to know who will be the possible sender, a concept "*sender set*" is defined first.

Given a guarded protocol $\tau = (T, s, F, M, C, \delta)$, the *sender set* of a state $t \in T$, denoted as $sdr(t)$, is a set containing all artifact interfaces α that is the sender of m where $(t, c, m, t') \in \delta$ for some $t' \in T$, $c \in C$, and $m \in M$. In Fig. 5(a), the sender sets for t_1 to t_6 are {`Order`, `Payment`}, {`Payment`}, {`Purchase`}, $\varnothing$, {`Order`}, and $\varnothing$, resp.

Sender sets are known at design time, the current sender can choose the next sender from the corresponding sender set of the current state at runtime. The initial sender should be delegated externally by, e.g., the environment. These steps are repeated until a peer (with the token) reaches a final state. This peer then informs all other peers to end the execution by sending messages of flag FIN. Alg. 1 accomplishes the coordinator that runs on individual peers.

Algorithm 1. Coordinator for Peer p

Input: sdr, $p = (T, s, F, M, C, \delta_s, \delta_r, \delta_\varepsilon)$

```
1:  loop
2:      Wait for the next message m
3:      if Flag(m) = SND then
4:          if ∃c ∈ C, ∃m' ∈ M, ∃t ∈ T,
            (State(m), c, m', t) ∈ δ_s then
5:              Send m' (flag: RCV, state: t);
6:              randomly select s from sdr(t);
7:              Send to s a sync message
                (flag: SND, state: t);
8:          end if
9:      else if Flag = RCV then
10:         if State(m) ∈ F then
11:             Boardcast sync message
                (flag: FIN, state: State(m))
12:             Terminate
13:         end if
14:     else
15:         Terminate {"FIN" case}
16:     end if
17: end loop
```

Example 47. The following describes a possible execution. The user chooses and sends to `Order` (in the sender set for t_1). `Order` sends a CP (flag: RCV) to `Purchase` and inform `Payment` to be the next sender (through a sync message with flag SND) since $sdr(t_2)$ = {`Payment`}. After `Payment` sends a CH to `Order`, it will pick `Purchase` from $sdr(t_3)$. Finally, `Purchase` sends a PC to `Fulfillment`. Once the `Fulfillment` reaches its final state t_4, FIN messages will be broadcast. ∎

Theorem 48. Given a guarded conversation protocol τ, each sequence of ground messages, is accepted by τ iff it is generated by Alg. 1 running for guarded peers of τ.

Remark 49. Denote by N^* the number of regular messages that should be sent, and $\hat{N}$ as the number of regular and synchronization messages sent according to Alg. 1. It is easy to see that $\hat{N} < 2N^* + (k - 1)$, where k is the number of peers. Furthermore, if FIN messages are not needed, the bound is reduced to $\hat{N}/N^* < 2$.

5 Related Work

The choreography approach to modeling and analysis BP or service interactions has been studied for a decade. A survey for formal models and results is provided in [21].

A WS-CDL choreography constrains message exchanges based on conditions that may involve information types, variables and tokens. Message contents need be copied to variables to be used in conditions. There is no data model for participants in a collaboration, nor direct support for multi-instances of a participant (type).

Let's Dance [24] provides a set of sequencing constraint primitives to allow a choreography to be specified in a graphical language. It lacks a clearly support for data or

information models. Earlier work on conversations was reported in [9]. In the conversation model, BP systems collaborate with each other via generic asynchronous message exchanging. The information model in the conversation is limited.

BSPL [19] models messages with input and output parameters and compose the messages into protocols where constraints on message sequences are derived from the input/out parameters such that each parameter is defined exactly once in execution. A realization mechanism is reported in [20].

Recent work in [6,12] extend BPEL to support choreographies with a bottom-up approach to build a choreography from specified participant behaviors. BPEL4chor supports service interaction patterns, e.g., one-to-many send, one-from-many receive, one-to-many send/receive patterns through aggregation. Similar work on BPMN was reported in [18]. However, neither BPEL nor BPMN's extensions directly include data in their conceptual models and instance level correlation support is much weaker.

Artifact-centric choreography [14] extends existing artifact-centric BP models with agents and locations. BPs can access artifacts from their locations with the help of agents. Petri-Net is used to specify artifact internal behaviors and external interactions. The model has no artifact data attributes.

There have been work done on testing if a choreography is realizable. In [15,16], a choreography is defined wrt a set of peers forming a collaboration. The notions of completed, partial and distributed realizability of choreography were defined and studied. It was shows that partial realizability is undecidable whereas distributed and complete realizability are decidable. [2,1] focused on the realizability problem of global behavior of interaction services. Sufficient conditions are given for realizability.

6 Conclusions

This paper proposes a declarative choreography language that 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. Furthermore, a subclass of the rule-based choreography is shown to be equivalent to a state-machine-based choreography.

References

1. Conversation protocols: a formalism for specification and verification of reactive electronic services. Theo. Comp. Sci. 328(1-2), 19–37 (2004)
2. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: a new approach to design and analysis of e-service composition. In: Proc. Int. Conf. on World Wide Web, WWW (2003)
3. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and Orchestration Conformance for System Design. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 63–81. Springer, Heidelberg (2006)
4. Carbone, M., Honda, K., Yoshida, N., Milner, R., Brown, G., Ross-Talbot, S.: A theoretical basis of communication-centred concurrent programming (2006)
5. Cattell, R., Barry, D.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann (2000)
6. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for Modeling Choreographies. In: Proc. 5th Int. Conf. on Web Services, ICWS (2007)

7. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: Proc. Int. Conf. on World Wide Web, WWW (2004)
8. Hammer, M., Champy, J.: Reengineering the Corporation: A Menifesto for Business Revolution. Harper Business Press, New York (1993)
9. Hanson, J., Nandi, P., Kumaran, S.: Conversation support for business process integration. In: Proc. Int. Conf. on Enterprise Distributed Object Computing, EDOC (2002)
10. Hull, R., Llirbat, F., Simon, E., Su, J., Dong, G., Kumar, B., Zhou, G.: Declarative Workflows that Support Easy Modification and Dynamic Browsing. In: Proc. Int. Joint Conf. on Work Activities Coordination and Collaboration, WACC (1999)
11. Katila, R., Mang, P.Y.: Exploiting technological oppurtonities: The timing of collaborations. Research Policy 32(2), 317–332 (2003)
12. Kopp, O., Engler, L., van Lessen, T., Leymann, F., Nitzsche, J.: Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus. In: Fleischmann, A., Schmidt, W., Singer, R., Seese, D. (eds.) S-BPM ONE 2010. CCIS, vol. 138, pp. 36–53. Springer, Heidelberg (2011)
13. Liu, C., Li, Q., Zhao, X.: Challenges and opportunities in collaborative business process management: Overview of recent advances and introduction to the special issue. Information Systems Frontiers 11(3), 201–209 (2009)
14. Lohmann, N., Wolf, K.: Artifact-Centric Choreographies. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 32–46. Springer, Heidelberg (2010)
15. Lohmann, N., Wolf, K.: Realizability Is Controllability. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 110–127. Springer, Heidelberg (2010)
16. Lohmann, N., Wolf, K.: Decidability Results for Choreography Realization. In: Kappel, G., Maamar, Z., Motahari-Nezhad, H.R. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 92–107. Springer, Heidelberg (2011)
17. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
18. Pfitzner, K., Decker, G., Kopp, O., Leymann, F.: Web Service Choreography Configurations for BPMN. In: Di Nitto, E., Ripeanu, M. (eds.) ICSOC 2007. LNCS, vol. 4907, pp. 401–412. Springer, Heidelberg (2009)
19. Singh, M.: Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language. In: Proc. Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS), pp. 491–498 (2011)
20. Singh, M.: LoST: Local state transfer. In: Proc. Int. Conf. on Web Services, ICWS (2011)
21. Su, J., Bultan, T., Fu, X., Zhao, X.: Towards a Theory of Web Service Choreographies. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 1–16. Springer, Heidelberg (2008)
22. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)
23. Vardi, M., Wolper, P.: Reasoning About Infinite Computations. Inf. Comput. 115(1) (1994)
24. Zaha, J.M., Barros, A., Dumas, M., ter Hofstede, A.: Let's Dance: A Language for Service Behavior Modeling. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 145–162. Springer, Heidelberg (2006)