From Visual Choreographies to Flexible Information Protocols

Tom Lichtenstein[1], Amit K. Chopra[2], Munindar P. Singh[3], and Mathias Weske[1]

[1] Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
tom.lichtenstein@hpi.de, mathias.weske@hpi.de
[2] Lancaster University, Lancaster, UK
amit.chopra@lancaster.ac.uk
[3] North Carolina State University, Raleigh, NC, USA
mpsingh@ncsu.edu

Abstract. Choreographies enable the coordination of interactions between business partners. Established modeling languages such as BPMN focus on a visual notation that may facilitate design but lacks formal semantics. Moreover, such notations encourage the explicit ordering of interactions, which often results in over-constrained models. In contrast, information protocols provide a precise and flexible operational model for interaction. This paper contributes a tool-supported, semi-automated mapping from object-aware BPMN choreography diagrams to information protocols. Our approach enables business experts to tailor the flexibility of the resulting protocols to their requirements.

Keywords: Business processes, Interaction protocols, Multiagent Systems, Messaging, Information flow

1 Introduction

This paper focuses on modeling interactions between the business processes of collaborating organizations. Current business process modeling languages such as BPMN choreography diagrams [19] capture coordination requirements in terms of ordering constraints on message exchanges. The visual nature and wide adoption are significant strengths of these languages [8]. However, these languages often rely on explicit interaction sequencing, which can lead to complex models or overly constrained behavior [17]. The latter may interfere with the underlying business requirements, especially in light of changes in internal needs or the external environment [21]. Nonetheless, distinguishing between intended and arbitrary constraints is nontrivial, complicating the creation of flexible yet precisely constrained models. In addition, despite progress on object-aware choreographies [14], current models remain non-operational due to a lack of essential details about the information transferred in each interaction. Thus, providing operational semantics could facilitate the implementation of interaction behaviors by providing a clear interface to the processes involved.

Contributions. Consequently, this paper addresses the following research question: *How can we derive flexible and operational models from object-aware choreographies?* To this end, we adopt the idea of *information protocols* specified in the *Blindingly Simple Protocol Language* (BSPL) [22,23]. Information protocols are declarative, have a formal semantics, can be enacted by decentralized agents representing the parties, and can support highly flexible interactions—setting them apart from other protocol languages [4]. Unlike traditional business process models, which explicitly specify the ordering of tasks and messages, typically via control-flow abstractions, information protocols specify causal dependencies and integrity constraints declaratively based on information, thus avoiding unnecessary control-flow constraints while preserving necessary data constraints.

Our main contribution is an approach for generating information protocols from object-aware choreographies that preserves business meaning. Figure 1 provides our key intuitions: given a choreography (a), we generate a protocol (b) while ignoring control-flow constraints, allowing for a more flexible ordering of the choreography's interactions. Since not all the resulting flexibility may be desired by the business partners, our method supports business experts in fine-tuning the protocol by applying a modified or reduced set of relevant control-flow constraints discovered from the choreography. Thus, the refined protocol (c) can be tailored to the needs of the collaborating organizations.

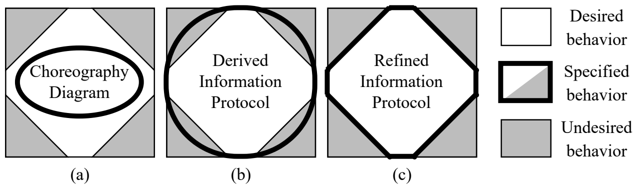

Fig. 1. The interaction behavior allowed by (a) a choreography, (b) a derived information protocol, and (c) a refined protocol, adapted from [17].

The paper is organized as follows: Section 2 introduces object-aware process choreographies and interaction protocols. Section 3 maps choreographies to BSPL. Section 4 identifies and integrates control-flow constraints. Section 5 evaluates our approach using two scenarios, and Section 6 reviews related work in the area. Finally, Section 7 concludes the paper and outlines directions for future research.

2 Preliminaries

We now introduce the core background for this paper.

2.1 Object-Aware Process Choreographies

Process choreographies define the ordering of message exchanges between collaborating *participants*. We focus on BPMN choreography diagrams, in the following referred to as *choreography*, as an interaction-oriented abstraction of BPMN collaboration diagrams [19]. A choreography comprises *choreography tasks*, each specifying a message sent by one participant to another (optionally with a response), as Figure 2 shows. This choreography begins with the buyer placing an order. Next, the shop sends an invoice. The buyer either cancels the order (ending the choreography) or pays. When the invoice is paid, the shop forwards the order to the warehouse. Once packed and approved, the warehouse ships the parcel, and the buyer's confirmation of receipt completes the choreography. Sequence flow defines order constraints between tasks, while gateways indicate exclusive and parallel behavior.

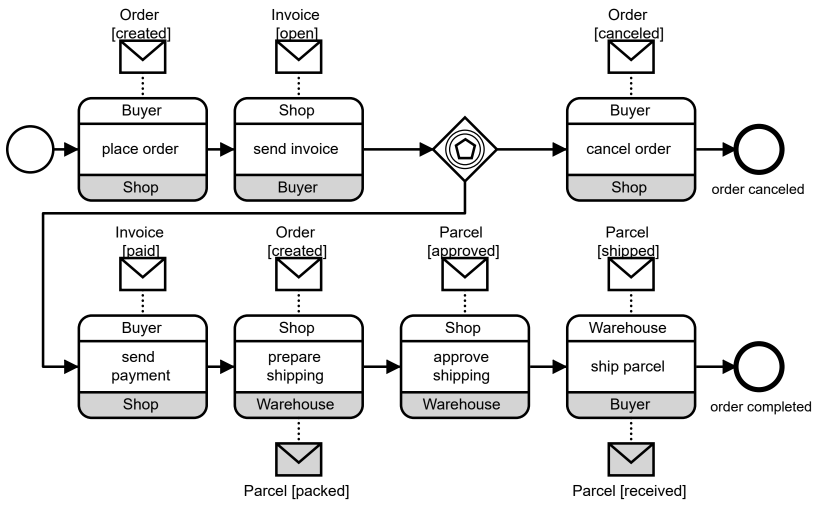

Fig. 2. A BPMN choreography diagram for an order management choreography between a buyer, a shop, and a warehouse. Participants in white bands initiate the respective task. Gray envelopes indicate response messages.

Lichtenstein et al. [14] extend choreographies by incorporating a *shared data model* and *shared object lifecycles*. Each class in the shared data model specifies a type of message that is used to annotate the message elements in a choreography. Figure 3 shows a shared data model for the choreography in Figure 2. The inclusion of attributes is discussed in Section 3.1. Relations between classes may impose ordering constraints on the creation of objects [14]. For example, according to Figure 3, creating an *Invoice* or *Parcel* requires an existing *Order*.

Objects' states are modified during execution. A shared object lifecycle defines the allowed states and state transitions for each class, as exemplified in Figure 4. State transitions without a source state indicate the creation of objects. In addition, arc inscriptions specify the participant that can perform the transition. For example, only a *Buyer* (B) can create an order and only a *Warehouse* (W) can change a parcel's state to *shipped*. States are changed locally and synchronized with participants via messages. In a choreography, message annotations in square

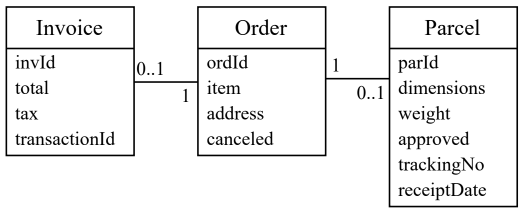

Fig. 3. A shared data model, as a UML class diagram, illustrating the structure of the data being exchanged for the order management choreography, including attributes.

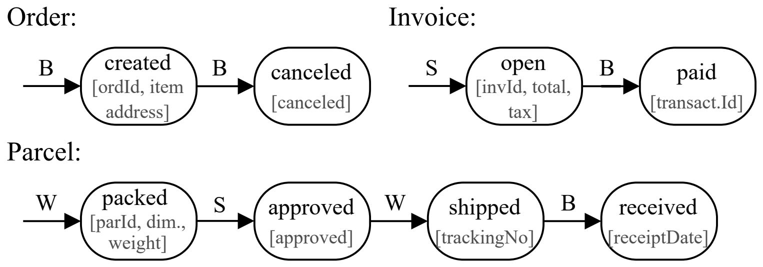

Fig. 4. Shared lifecycles for data objects of the classes *Order*, *Parcel*, and *Invoice* extended with attribute references. Participant names are abbreviated to initials.

brackets indicate the state in which an object must be sent [14]. Section 3.1 discusses the associations between states and attributes shown in Figure 4.

2.2 Information Protocols

Information protocols capture the ordering and occurrence of messages between autonomous parties while neglecting their internal reasoning [7]. The Blindingly Simple Protocol Language (BSPL) is an operational, declarative language to capture possible interactions based on the information available to the agents [22]. Listing 1 shows an example BSPL *protocol*. Each protocol consists of a name (Line 1), a set of roles (Line 2), a public (Line 3) and private (Line 4) set of parameters representing the units of information exchanged, and message schemas (Lines 6–9) or references to other protocols (Line 10) allowing composition. At runtime, each business partner would adopt a role. Message schemas define a sender, a receiver, and the parameters exchanged. In a protocol enactment, a parameter is either bound to a value (available for reading but not for writing) or unbound. An enactment is identified by a binding of the public *key* parameters.

For each interaction, *in*, *out*, and *nil* adornments specify requirements on the availability of the parameters. Parameters adorned with *in* must be bound to enable the interaction, e.g., in Listing 1, *ship* can be sent only if the *orderId* is bound. An *out* adornment binds an initially unbound parameter once the message is sent, e.g., sending an *order* message binds the parameters *orderId*, *item*, and *address* to their respective values. Finally, *nil* requires the parameter to be unbound for execution. Hence, *outcome* must not be bound to send *req_cancel*.

```
OrderManagement {
  roles B, S
  parameters out orderId key, out item, out outcome
  private address, price, receipt, rescind

  B→S: order  [out orderId, out item, out address]
  S→B: ship   [in orderId, in item, in address, in receipt, nil rescind,
     out outcome]
  B→S: req_cancel [in orderId, nil outcome, out rescind]
  S→B: ack_cancel [in orderId, in rescind, out outcome]
  Payment(S, B, in orderId, in item, nil outcome, out price, out receipt)
}
```

Listing 1. Order management protocol adapted from [22].

Notably, *OrderManagement* supports concurrent sending of *cancel* and *ship* messages by different agents, with the shipping preventing *ack_cancel* by binding *outcome*. Traditional protocol languages based on communicating state machines do not allow for such concurrency [4]. Nonetheless, a protocol should never allow a parameter to be bound by multiple agents concurrently to ensure *safety*, which is crucial for operability and can be statically verified [25]. Therefore, prioritizing shipping in Listing 1 is essential to support concurrency without violating safety.

3 From Choreographies to Protocols

We now describe a semi-automatic approach to mapping object-aware choreographies to BSPL, using the example from Section 2.1. The supplementary models are first enriched with attributes and then mapped to a BSPL protocol.

3.1 Extending Object-Aware Choreographies with Attributes

Object-aware choreographies provide a high-level view of data using objects and states, whereas protocols rely on low-level parameters to specify the information to be exchanged. To bridge the gap between these levels of abstraction, we enrich shared data models and lifecycles with attributes. Each attribute represents a unit of information associated with an object that is exchanged via messages. Therefore, for the mapping, we interpret attributes as parameters. As depicted in Figure 3, each class is associated with a set of attributes. Inspired by the work of Pérez-Álvarez et al. [20], we interpret the state of an object as a binding of attributes. The mapping of binding to state is defined in the corresponding object lifecycle (cf. Figure 4). Binding the respective attributes to a value changes the object's state accordingly. For example, an invoice in state *open* has only *invId*, *total*, and *tax* bound. Transitioning to *paid* requires binding *transactionId* as well. To avoid ambiguity, each state must be associated with a unique binding.

3.2 Mapping Object-Aware Choreographies to Protocols

A protocol defines the legal interactions in a choreography. Hence, our mapping uses the choreography tasks and the enriched shared data model and shared object lifecycles as input. Since BSPL supports composition [22], we create a *class* *protocol* for each class of the data model and compose them into the resulting protocol. A class protocol defines roles, parameters, and message schemas related to one class. Listing 2 illustrates the class protocol generated for the class *Order*. The mapping rules for class protocols and their composition are detailed below.

```
Order {
  roles B, S, W
  parameters out ordId key, out item, out address, out canceled
  private fw_created

  B→S: create    [out ordId, out item, out address]
  S→W: fw_create [in ordId, in item, in address, nil canceled, out fw_created]
  B→S: cancel    [in ordId, in item, in address, out canceled]
}
```

Listing 2. *Order* class protocol.

Roles and Public Parameters. The roles are inferred from the participants involved in choreography tasks associated with the class. The class attributes of the shared data model serve as public parameters. These parameters are adorned *out*, as they are only bound by the respective class protocol. At least one class attribute must be selected as a key parameter for a protocol. Each key parameter must reflect an attribute associated with all possible initial states in the lifecycle. For our running example, we select *ordId*, *invId*, and *parId* as the key parameters for the respective class protocols. A class protocol may require key parameters of other class protocols as *in* parameters to express dependencies according to the shared data model [14]. Referencing other key parameters is required if there is a relation to another class with a multiplicity having a lower-bound greater than zero, as for *Invoice* and *Parcel* in Figure 3. Consequently, as Listings 3 and 4 show, both class protocols take the key parameter *ordId* of the *Order* protocol as *in*. For one-to-one relations, the creation of both objects is merged into one message schema that is added to both class protocols, including the respective adaptation of the public parameters. For brevity, we do not discuss verifying upper bounds greater than one, as this requires more advanced language features [3].

```
Invoice {
  roles S, B
  parameters in ordId key, out invId key, out total, out tax, out transactionId

  S→B: open [in ordId, out invId, out total, out tax]
  B→S: pay  [in invId, in total, in tax, out transactionId]
}
```

Listing 3. *Invoice* class protocol.

```
Parcel {
  roles W, S, B
  parameters in ordId key, out parId key, out dimensions, out weight,
     out approved, out trackingNo, out receiptDate

  W→S: pack    [in ordId, out parId, out dimensions, out weight]
  S→W: approve [in parId, in dimensions, in weight, out approved]
  W→B: ship    [in parId, in approved, out trackingNo]
  B→W: receive [in parId, in trackingNo, out receiptDate]
}
```

Listing 4. *Parcel* class protocol.

Message Schemas. A class protocol defines message schemas based on the tasks associated with the class in the choreography, relations to other classes, and the state transitions in the shared object lifecycle. Figure 5 shows the message schemas for the *Invoice* protocol based on the tasks *send invoice* and *send payment*. For the mapping, we interpret two-way tasks as two sequential one-way tasks with alternating initiators [6]. Given a task with a class and a state, a message schema is created for each transition towards the state that can be performed by the task's initiator. For example, for *send payment*, the buyer can perform one state transition towards *paid* according to the lifecycle in Figure 5. Hence, one message schema is added to the class protocol for this task.

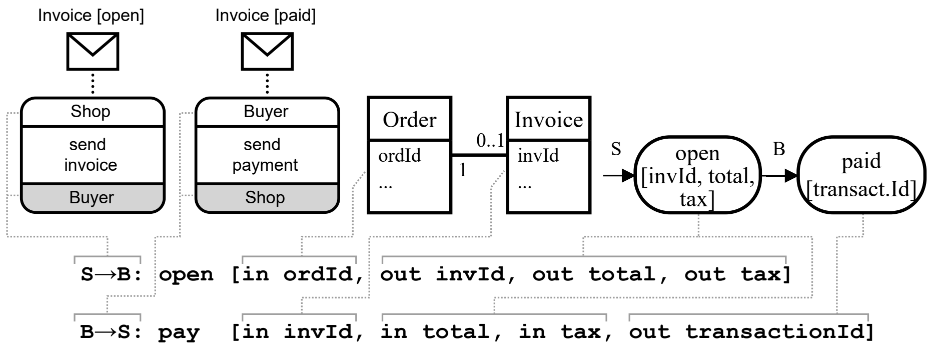

Fig. 5. Mapping rules for the message schemas of the *Invoice* class protocol.

Each message schema is named according to the state in which the task sends the object in active form. For example, the message sending an invoice in state *paid* is named *pay*. Sender and receiver are adopted from the task. The parameters of each message result from the state transition associated with the message schema. Each message schema specifies the parameters associated with the target state as *out*, e.g., *transactionId* for *pay*. Since *out* parameters can be bound at most once per enactment, only acyclic lifecycles are supported.

If a source state exists, the parameters associated with the source state and the key parameters of the class protocol are added as *in* parameters, e.g., *invId*, *total*, and *tax* (cf. Figure 5). If there are alternative transitions originating from the source of the transition associated with the message, all parameters of the alternative target states must be added as *nil* parameters to enforce exclusivity. If no source state exists, the key parameters of the class protocol are adorned *out* and the key parameters of the classes on which the considered class depends are added as *in* parameter. Considering the message *open* for the task *send invoice* in Figure 5, *ordId* is adorned *in* to enforce a relation to an existing order.

Whereas most messages add information to the choreography, some only forward already exchanged information, e.g., as *prepare shipping* forwards an order to the warehouse (cf. Figure 2). A forwarding message schema takes all parameters associated with the required state as *in* and all parameters associated with directly succeeding states as *nil*, ensuring the object is in the correct state when forwarded. A private *out* parameter is added to capture that a forwarding took place. A forwarding message schema is added when (1) the initiator of a task cannot transition to the required state, or (2) multiple tasks can send the object in the same state, all of which have an initiator that can transition to that state. In the latter case, it is not clear which message performs the transition, so that forwarding messages are inserted in addition to the transitioning messages. Since a task can now be associated with both a transitioning and a forwarding message, both message schemas must contain the same private *out* parameter so that only one can be sent. Listing 2 shows an example of a forwarding message schema. For clarity, forwarding schemas are prefixed with *fw*.

Composition. Listing 5 shows the result of composing the class protocols into one. Some parameters are omitted due to space constraints.

```
RelaxedOrderManagement {
  roles B, S, W
  parameters out ordId key, out invId key, out parId key, out item, ...

  Order(B, S, W, out ordId, out item, out address, out canceled)
  Invoice(S, B, in ordId, out invId, out total, out tax, out transactionId)
  Parcel(W, S, B, in ordId, out parId, out dimensions, out weight, ...)
}
```

Listing 5. Relaxed order management protocol combining all class protocols.

4 Fine-Tuning Flexibility

The derived protocol may be more flexible than the original choreography, since it incorporates only the data-related constraints. However, as illustrated in Figure 1(b), the protocol may allow undesirable behavior that the choreography avoids via control-flow constraints. For example, Listing 5 allows the following sequence of interactions: ⟨create, fw_create, pack, authorize, ship, receive, open, pay, cancel⟩, which permits shipping before payment and cancellation after shipping. This behavior may not be acceptable in a business context. We therefore propose an approach to identify relevant control-flow constraints from the initial choreography that can be used to constrain the derived protocol's behavior. By modifying or removing the identified constraints, the protocol can be refined according to business needs, as Figure 1(c) shows.

4.1 Identifying Control-Flow Constraints

Given a protocol and a choreography, we derive constraints by detecting interaction sequences of a protocol that deviate from a given choreography. For each deviation, we extract the violated control-flow constraints from the choreography as output, as described in Algorithm 1. Lines 3–4 determine all possible sequences of interactions assuming synchronous communication, i.e., sent messages are received before the next message is sent. Lines 5–6 filter all interaction sequences produced by the protocol that are not supported by the choreography. For each filtered sequence, the first message that deviates from the behavior of the choreography is identified (Line 7) and the deviating message is associated with its corresponding task to derive the applicable control-flow constraints (Line 8). Lines 10–11 add constraints that are violated by the sequence to the output.

Algorithm 1: Identifying protocol constraints given a choreography.

```
   Input: $p$ (protocol), $c$ (choreography)
   Output: $\mathcal{C}$ (set of constraints)
1  Function identify_control_flow_constraints($p, c$):
2      $\mathcal{C} \leftarrow \emptyset$; // Set of discovered constraints
3      $S_p \leftarrow$ determine_interaction_sequences($p$);
4      $S_c \leftarrow$ determine_interaction_sequences($c$);
5      for $\sigma_p \in S_p$ do
6          if $\sigma_p \notin S_c$ then
7              $m \leftarrow$ get_deviating_message($\sigma_p, S_c$);
8              $\mathcal{C}_t \leftarrow$ infer_constraints(get_task($m, c$), $c$);
9              for $k \in \mathcal{C}_t$ do
10                 if sequence_violates_constraint($k, m, \sigma_p$) then
11                     $\mathcal{C} \leftarrow \mathcal{C} \cup \{k\}$;
12     return $\mathcal{C}$;
```

We distinguish two types of control-flow constraints. A *precedence* constraint requires that the preceding task always occurs before the succeeding task. An *exclusion* constraint between the two tasks implies that no interaction sequence contains both. If the sequence flow arc connects two tasks, a precedence from the source to the target is identified, e.g., in Figure 2, *send payment* precedes *prepare shipping*. In case the arc connects the observed task to a gateway, precedence is derived from the observed task to all tasks following the gateway. Similarly, if an arc connects a gateway to the observed task, precedence is imposed from the preceding tasks to the observed task; e.g., *send invoice* precedes *send payment*. To ensure that precedence constraints can be enforced, we only consider realizable choreographies [9]. In addition, if an exclusive or event-based gateway is connected to the observed task, exclusion is derived for all tasks connected to the gateway on exclusive paths, e.g., *cancel order* excludes *send payment* and vice versa.

Finally, all discovered constraints that are violated by the interaction sequence up to the deviating message are added to the output, essentially removing the interaction sequence from the protocol to achieve trace-based conformity with the choreography [6]. In case multiple precedences share the same target with the sources being exclusive, only one of the precedences must hold. Figure 6(a) illustrates the constraints discovered from Listing 5 and Figure 2. Though the discovered control-flow constraints serve as a baseline for the refinement of the protocol, constraints might be modified or omitted by experts in accordance

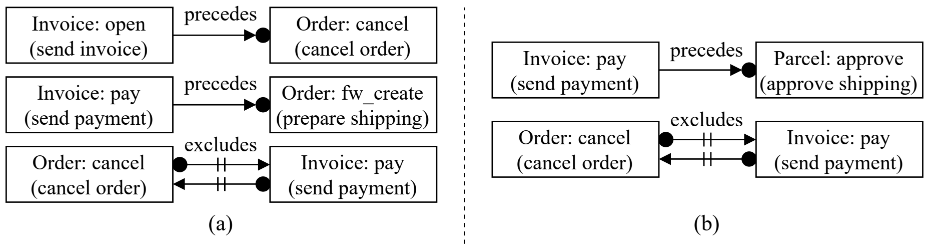

Fig. 6. Control-flow constraints for the protocol in Listing 5, in the notation of [17]: (a) derived from the control flow of Figure 2 and (b) a possible relaxation of the constraints. Messages are associated with the respective class protocol name and task label.

with business needs. Figure 6(b) shows a possible relaxation that allows for preparing the parcel before receiving payment: ⟨create, open, fw_create, pack, pay, authorize, ship, receive⟩. Hence, the relaxed constraints allow for behavior beyond the choreography, while ensuring that the invoice is paid before a parcel is shipped and that a shipped order cannot be canceled (nor a canceled order shipped). In general, the realizability of the new constraints is crucial, but its verification is out of our present scope.

4.2 Refining Protocols

We can refine a protocol to satisfy additional control-flow constraints. In essence, the protocol is extended by parameters that are used to implement precedence and exclusion constraints to control message ordering. Specifically, precedence translates to an *out* parameter from the preceding message included as *in* in the succeeding message. Similarly, exclusion is achieved by adding the same *out* parameter to the exclusive messages. If multiple precede constraints share the same target, with the sources being exclusive, the target message is copied for all possible preceding messages. Each copy takes a parameter bound by a different preceding message as *in* parameter.

```
RefinedOrder {
  roles B, S, W
  parameters ..., in cf_p1, in cf_p2, out cf_e

  B→S: create    [...]
  S→W: fw_create [..., in cf_p1]
  B→S: cancel    [..., in cf_p2, out cf_e]
}

RefinedInvoice {
  roles S, B
  parameters ..., out cf_p1, out cf_p2, out cf_e

  S→B: open [..., out cf_p2]
  B→S: pay  [..., out cf_p1, out cf_e]
}
```

Listing 6. Refinement of the protocols in Listings 2 and 3 based on the constraints illustrated in Figure 6. Unchanged parameters are omitted.

Listing 6 refines Listings 2 and 3 based on the constraints in Figure 6(a). Here, cf_p1 ensures that ⟨pay⟩ precedes ⟨fw_create⟩, *cf_p2* guarantees that ⟨open⟩ precedes ⟨cancel⟩, and *cf_e* enforces exclusivity between ⟨pay⟩ and ⟨cancel⟩, thus implementing the constraints.

5 Evaluation

We evaluate the conformance, flexibility, and operability of protocols derived from object-aware choreographies to answer the research question presented in Section 1 based on two scenarios: S_1, an *Order Management* choreography described in Section 2.1, and S_2, a *Transport of Goods* choreography illustrated in Figure 7.

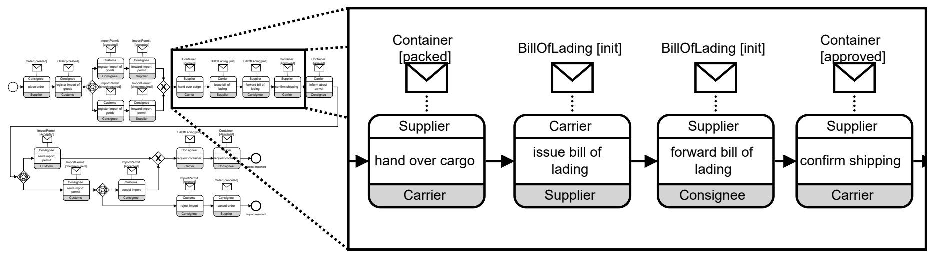

Fig. 7. Choreography adapted from [14], describing the interplay between a consignee, a supplier, a carrier, and customs to organize the transportation of goods. The model consists of 18 tasks and 5 gateways and 4 shared data model classes.

We adopt the number of distinct interaction sequences that can be generated by a given model as the primary metric for evaluating flexibility [5]. Furthermore, we compare the behavior of models and protocols using trace-based conformance [6]. In this context, we consider polymorphic message schemas to represent the same interaction. Only interaction sequences of a single protocol enactment are considered, assuming synchronous communication. Therefore, we evaluate conformance and flexibility according to three criteria:

- C_1 **Fitness**: Derived protocols cover the entire behavior of the choreography.
- C_2 **Precision**: Protocols, when refined with control-flow constraints discovered from the choreography, match the behavior of the choreography.
- C_3 **Relaxation**: Refined protocols with fewer control-flow constraints allow for more distinct interaction sequences.

The first two criteria are adapted from the quality dimensions for process model discovery [2], while the third criterion assesses the degree of flexibility achieved by applying only a subset of discovered constraints [5]. To evaluate these criteria for the given scenarios, we developed a prototype to automate the mapping from choreographies to protocols, identify control-flow constraints, and create refined protocols. For S_1, we identified three constraints (two precedence and one exclusion), as shown in Figure 6(a). For S_2, we identified 15 constraints (13 precedence and two exclusion). Based on the results, we derive three types of refined protocols for each scenario: (1) protocols containing only exclusion constraints, (2) protocols containing only precedence constraints, and (3) protocols containing both types of constraints. For each resulting protocol, we compute the number of distinct interaction sequences. The results are presented in Table 1.

Table 1. Distinct interaction sequences computed for both scenarios using choreographies, derived protocols, and refined protocols including (1) exclusion constraints only, (2) precedence constraints only, and (3) both types of discovered constraints.

			Refined Protocol		
Scenario	**Choreography**	**Protocol**	**Exclusion**	**Precedence**	**Both**
S_1	2	143	58	7	2
S_2	3	11,699,340	4,948,100	49	3

Our results show that for both scenarios, the derived protocols support significantly more interaction sequences than the original choreographies. We reason that the large number of tasks relative to the number of classes is responsible for the significant increase in interaction sequences for S_2. Since tasks operating on objects of different classes can act independently after object creation, S_2 allows for high concurrency, leading to the comparatively large number of different interaction sequences without control-flow constraints.

Furthermore, the interaction sequences derived from refined protocols represent a subset of the parent protocol's sequences. When all identified constraints are applied, the protocols exactly match the interaction sequences of the choreographies, thus confirming C_1 and C_2. In addition, removing exclusion, precedence or both types of constraints increases the behavioral flexibility of the protocols, confirming C_3.

By using tooling from [25], we verified that all derived protocols comply with BSPL syntax and are safe. In conclusion, our approach effectively derives flexible, operational models from object-aware choreographies, as demonstrated by the results for both evaluation scenarios.

Limitations. Despite promising results, this approach relies on consistency between the choreography, the shared data model, and the shared object lifecycle to form valid protocols. Conflicting models can lead to deadlocks. States currently require a unique set of attributes for clear separation, which may lead to artificial attributes that provide no value besides identifying the state. In addition, we associate one choreography instance with one protocol enactment, which prevents support for loops and many-to-one relationships between data objects. While

BSPL allows expressing iterative behavior using multiple key bindings, relaxing this assumption requires further investigation.

When refining protocols, mapping each constraint to a parameter increases protocol complexity. Optimizing parameter introduction, such as reusing parameters for multiple constraints, could reduce protocol size. Furthermore, the relaxation of constraints is up to the end user. The approach could benefit from a methodology to guide the relaxation. Visual representations of protocol behavior could aid business experts in assessing the resulting protocols.

In addition, our discovery of control-flow constraints and evaluation assumes synchronous communication for computing possible interaction sequences. Given that protocols support asynchronous communication, the flexibility gains and limitations that asynchrony implies need further investigation. Finally, the evaluation is limited to two scenarios. Further validation with more complex scenarios, including consideration of applicability by business experts, would substantiate the results.

6 Related Work

Data-aware and declarative choreographies have garnered much attention. Knuplesch et al. [13] model interorganizational data exchange by extending choreographies with virtual data objects that act as variables for routing conditions. Similarly, Meyer et al. [16] enhance BPMN collaboration diagrams with a global data model to automate data exchange and transformation between global and local data. Adding to this, Nikaj et al. [18] incorporate RESTful specifications to coordinate data exchange in choreographies. Whereas these works model data exchange, they rely on explicit interaction ordering, limiting flexibility.

Montali et al. [17] propose DecSerFlow, a declarative language that uses linear temporal logic to constrain message ordering and logical expressions for data constraints. Building on this, Sun et al. [26] introduce artifact-centric choreographies, which treat interacting processes as artifacts that are accessed and manipulated via messages. Geatti et al. [10] extend DECLARE for collaborative processes, partitioning constraints into assumptions for external participants and guarantees for the local process using LTL on finite traces. Expanding further, Hildebrandt et al. [12] present a formal model for declarative choreographies based on dynamic condition response graphs, later extended with temporal and data constraints in [11]. While the approaches achieve flexibility through declarative constraints, they still rely on explicit message ordering. Since our approach aims to relax control-flow constraints while preserving data constraints, we chose BSPL as our target language. BSPL's inherent focus on specifying information dependencies allows us to naturally encode data dependencies without the need to infer explicit message ordering.

Bergman et al. [1] discover declarative control-flow constraints from BPMN process models to be used for conformance checking. Contrary to their work, we aim to discover control-flow constraints that affect flexibility in a distributed scenario, allowing for fine-granular relaxation of constraints. Meroni et al. [15]

map BPMN process diagrams to more flexible E-GSM models for artifact-based monitoring of multi-party processes. Nonetheless, their approach abstracts from interaction behavior among collaborators. Finally, Singh [24] introduces Bliss, an extension of BSPL, providing a systematic methodology for specifying information protocols. The methodology iteratively identifies the required information to produce protocol artifacts, ensuring flexibility and avoiding over-constrained specifications. In contrast, our approach infers protocols from visual object-aware choreographies.

7 Conclusion

We propose a novel approach that maps object-aware BPMN choreography diagrams to BSPL information protocols to address the need for flexibility and operability in modeling collaborations between autonomous organizations. By initially neglecting control-flow constraints, the resulting protocol allows for more flexible behavior. To prevent undesirable behavior, constraints are identified that limit the protocol to the choreography specification. By selectively relaxing these constraints, business experts can refine the protocol's behavior to increase flexibility while avoiding undesired behavior. Evaluation shows that protocols refined with the identified constraints match the behavior of the original choreography, while removing constraints increases flexibility. Furthermore, all generated protocols are safe and adhere to BSPL syntax, validating operability.

Future research includes adding support for loops and investigating the impact of asynchronous communication on flexibility and conformance with the original choreography. Further enhancements include the development of constraint relaxation guidelines and visual representations of the resulting protocols to assist business experts in protocol design. Finally, user studies could validate the applicability of the approach.

Resources and Reproducibility

The source code of the prototype, all resources used for the evaluation, and a screencast demonstrating the prototype are available on GitHub[1].

Acknowledgements

This work was supported by DFG grant 450612067 (TL), EPSRC grant EP/N027965/1 (AKC), and NSF grant IIS-1908374 and a gift from SAS Institute (MPS).

References

1. Bergmann, A., Rebmann, A., Kampik, T.: BPMN2Constraints: Breaking down BPMN diagrams into declarative process query constraints. In: BPM Demonstration & Resources Forum. CEUR-WS.org (2023)

[1] https://github.com/bptlab/chor2bspl

2. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: On the role of fitness, precision, generalization and simplicity in process discovery. In: OTM. LNCS, Springer (2012)
3. Chopra, A.K., Christie, S., Singh, M.P.: Splee: A declarative information-based language for multiagent interaction protocols. In: AAMAS. ACM (2017)
4. Chopra, A.K., Christie V, S.H., Singh, M.P.: An evaluation of communication protocol languages for engineering multiagent systems. JAIR (2020)
5. Corea, C., Felli, P., Montali, M., Patrizi, F.: On the flexibility of declarative process specifications. In: CAiSE. LNCS, Springer (2024)
6. Corradini, F., et al.: Collaboration vs. choreography conformance in BPMN. Log. Methods Comput. Sci. (2020)
7. Desai, N., Mallya, A.U., Chopra, A.K., Singh, M.P.: Interaction protocols as design abstractions for business processes. IEEE Trans. Software Eng. (2005)
8. Dumas, M., Pfahl, D.: Modeling software processes using BPMN: When and when not? In: Managing Software Process Evolution. Springer (2016)
9. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. Theor. Comput. Sci. (2004)
10. Geatti, L., Montali, M., Rivkin, A.: Foundations of collaborative DECLARE. In: BPM Forum. LNBIP, Springer (2023)
11. Hildebrandt, T.T., López, H.A., Slaats, T.: Declarative choreographies with time and data. In: BPM Forum. LNBIP, Springer (2023)
12. Hildebrandt, T.T., Slaats, T., López, H.A., Debois, S., Carbone, M.: Declarative choreographies and liveness. In: FORTE. LNCS, Springer (2019)
13. Knuplesch, D., Pryss, R., Reichert, M.: Data-aware interaction in distributed and collaborative workflows: Modeling, semantics, correctness. In: CollaborateCom. IEEE (2012)
14. Lichtenstein, T., Weske, M.: Execution semantics for process choreographies with data. In: BPM Forum. LNBIP, Springer (2023)
15. Meroni, G., Baresi, L., Montali, M., Plebani, P.: Multi-party business process compliance monitoring through IoT-enabled artifacts. Inf. Syst. (2018)
16. Meyer, A., Pufahl, L., Batoulis, K., Fahland, D., Weske, M.: Automating data exchange in process choreographies. Inf. Syst. (2015)
17. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographiess. ACM Trans. Web (2010)
18. Nikaj, A., Weske, M.: Formal specification of restful choreography properties. In: ICWE. LNCS, Springer (2016)
19. OMG: Business Process Model and Notation (BPMN), V 2.0.2: Standard (2014)
20. Pérez-Álvarez, J.M., et al.: Verifying the manipulation of data objects according to business process and data models. Knowl. Inf. Syst. (2020)
21. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems – Challenges, Methods, Technologies. Springer, Berlin Heidelberg (2012)
22. Singh, M.P.: Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language. In: AAMAS. IFAAMAS (2011)
23. Singh, M.P.: Semantics and verification of information-based protocols. In: AAMAS. IFAAMAS (2012)
24. Singh, M.P.: Bliss: Specifying declarative service protocols. In: Proc. SCC (2014)
25. Singh, M.P., Christie V., S.H.: Tango: Declarative semantics for multiagent communication protocols. In: IJCAI. ijcai.org (2021)
26. Sun, Y., Xu, W., Su, J.: Declarative choreographies for artifacts. In: ICSOC. LNCS, Springer (2012)