Many-to-Many: Some Observations on Interactions in Artifact Choreographies

Dirk Fahland, Massimiliano de Leoni,
Boudewijn F. van Dongen, and Wil M.P. van der Aalst

Eindhoven University of Technology, The Netherlands
(d.fahland|m.d.leoni|b.f.v.dongen)@tue.nl, w.m.p.v.d.aalst@tm.tue.nl

Abstract. *Artifacts* have been proposed as basic building blocks for complex processes that are driven by life-cycle aware data objects. An *artifact choreography* describes the interplay of several artifacts from which the process "emerges". By design, an artifact choreography is tightly coupled to the process' underlying data model which gives rise to complex interactions between artifacts. This paper presents a simple model for these interactions and outlines open challenges in artifact choreographies.

Keywords: artifacts, choreography, interaction, synchronous, asynchronous

1 Introduction

The *artifact-centric* approach emerged in the last years as an alternative approach for precisely describing complex inter-organizational processes in a modular way [1–4]. The approach assumes that a process is driven by its data objects, called *artifacts*. Each artifact has its own *life-cycle* and can interact with other artifacts. In a service-oriented setting, each artifact's state can be updated by other artifacts via a well-defined *interface*, and the entire inter-organizational process follows from a *choreography* of its artifacts [3, 4].

What pushes artifact-centric choreographies beyond service choreographies is their tight coupling to the process' underlying data model. A process typically exhibits *many-to-many relationships* between its different data objects. For example, an order at an online-shop may be delivered in several packages where each package is delivered in a different truck. In turn, each truck usually delivers several packages of different orders. An artifact choreography inherits these many-to-many relations as a first-class concept: it describes how *several instances* of one artifact (e.g., order) interact with *several instances* of another artifact (e.g., deliveries).

This paper is devoted to explaining the subject of many-to-many relationships in artifact choreographies in more detail, and to highlighting specific challenges that arise in this setting. Using an instructive example, we present in Sect. 2 a *minimal* extension of service models that expresses *cardinality constraints* between artifact instances. This simple extension yields behavioral phenomena that only arise in the artifact-centric setting. We study these phenomena in Sect. 3 and note that a complete artifact choreography also must describe *stateful*

interaction protocol between artifact instances, and *which* instances interact with each other. We then show that the interaction between artifact instances can itself be expressed as a meaningful *coordinating artifact* that becomes part of the choreography. We conclude the paper in Sect. 4 by outlining two open research problems: (1) an automated generation of coordinating artifacts, and (2) ways to fully specify dynamic synchronization of artifact instances.

2 The Artifact-Centric Approach

The artifact-centric approach [1, 2] aims at a "more natural" approach of describing complex inter-organizational processes. Any process materializes itself in the artifacts (i.e., objects) that are involved in the process, and the artifacts' states. Examples of artifacts are a paper form, an electronic order, a package, or a delivery truck. State changes of an artifact usually follow a specific *life-cycle*: an artifact is *instantiated*; the state of an instance *changes* only via actions provided by the artifact; each artifact instance eventually reaches a *goal state* (e.g., a form gets signed, or a package is delivered). The key idea of the artifact-centric approach is that by just describing the artifacts' life-cycles and relations between artifacts, the process simply "emerges" from interactions of its artifact instances.

To better understand the subject, we consider the following example of an online shop's delivery process driven by 2 artifacts: *order* and *delivery* tour. The shop splits each order into several packages based on the availability of the ordered items. Several packages from different orders are then delivered in one tour. In case a package cannot be delivered, it is scheduled for another delivery tour or returned to the shop as undeliverable. The order is billed to the customer once all packages are processed. This behavior can be described by (1) for each artifact

We formally describe this process in an *artifact choreography* by describing the life-cycles of the artifacts *order* and *delivery* and how instances of these interact with each other. Many techniques are available for this purpose [5, 2–4]. Here, we employ *proclets* [5] as a formal model. Proclets minimally extend operational service models, e.g., [4], with *cardinality constraints* to express relations between artifact instances.

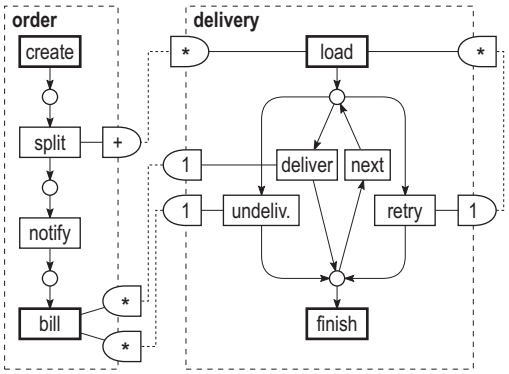

Fig. 1. An artifact choreography of a delivery process where orders can be split into multiple deliveries that can be retried and that can fail.

Artifact life-cycles, ports, and choreographies. To begin with, one proclet describes the life-cycle of one artifact as a *Petri net*. Figure 1 shows the life-cycles of an *order* and of a *delivery* tour inside the respective dashed boxes; the additional modeling

elements will be explained subsequently. A customer creates an order that is split into several packages by the availability of the ordered items; the order completes by notifying the customer about the order and sending the bill. A delivery tour begins with loading a delivery truck with all packages of the tour: each package is delivered, rescheduled for another another tour (retry), or declared as undeliverable before the next package is processed, until the tour finishes.

As the overall process follows from an *interaction* of orders and delivery tours, each proclet exposes some of its actions to other proclets via a *port*. A *proclet choreography* defines *channels* between proclet ports which describe how proclet instances interact with each other, e.g., by exchanging messages. The decisive difference to a service choreography comes by port *annotations* $(1, +, *)$ which specify how many messages an action sends to or receives from other proclet instances.

Formally, a proclet is a Petri net extended by ports; a choreography is a set of proclets with channels between ports.

Definition 1 (Proclet). *A* proclet $P = (N, ports)$ *is a Petri net* $N = (S, T, F)$ *with ports* $\subseteq 2^T \times \{in, out\} \times \{1, *, +\}$ *where each port* $p = (T_p, dir, card)$

1. *is associated to a set* $T_p \subseteq T$ *of transitions;*
2. *has a direction of communication (in: incoming port, the associated transitions receive a message, out: outgoing port, the associated transitions send a message);*
3. *has a* cardinality *card* $\in \{1, *, +\}$ *specifying how many messages may or have to be sent or received upon an occurrence of one* $t \in T_p$.

Definition 2 (Artifact choreography).
An artifact choreography $(\{P_1, \ldots, P_n\}, C)$ *consists of a finite set* $\{P_1, \ldots, P_n\}$ *of proclets together with a set* C *of channels s.t. each channel* $(p, q) \in C$ *is a pair of ports* $p, q \in \bigcup_{i=1}^{n} \text{ports}_i$ *with direction of* p *being in and direction of* q *being out.*

In Fig. 1, a half-round shape denotes a port and a dashed line a channel between two ports, e.g., there is a channel from split to load.

Artifact instances. During a process execution, artifacts of the choreography are *instantiated* and instances change their states according to the artifact lifecycle. We generally assume that each proclet P has a unique transition with an empty pre-set (no incoming arcs), and a unique transition with an empty post-set, which describe the creation and termination of instances, respectively. For example, an occurrence of create instantiates a new order of Fig. 1, an occurrence of bill terminates the instance; an instance of delivery is created by load and terminated by finish.

Data model and cardinality constraints. Yet, the notion of an artifact *instance* is much more crucial in artifact choreographies than in service choreographies. The artifacts describe the objects that drive the process. The process' *underlying data model* determines how many instances of one artifact (e.g., *order*)

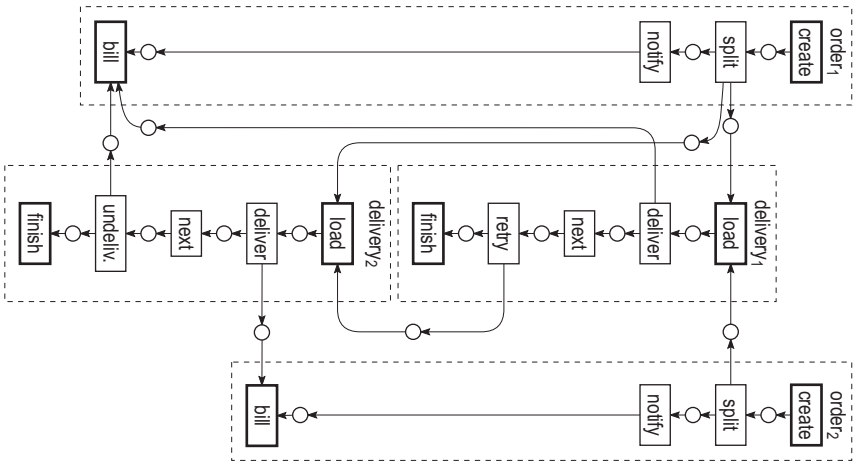

Fig. 2. A partially ordered run of the artifact choreography of Fig. 1.

may or must be related to how many instances of another artifact (e.g., *delivery*). For example, each *order* is delivered in one or more *delivery* tours (because it can be split), each *delivery* tour handles packages of several *orders*, and a delivery attempt of a tour can have a *follow-up* delivery tour.

In the artifact-centric setting, the process is driven by its artifacts. Hence, any two artifact instances that are related to each other also have to *interact* as the process evolves. The decisive contribution by proclets [5] is to incorporate this underlying data model of the process in the interaction specification. The *annotations* $(1, +, *)$ at the source and target ports of a channel from proclet A to proclet B specify how many instances of A interact with how many instances of B via the channel. This way, the port annotations at a channel define *cardinality constraints* on artifact instances.

For example, an order instance is split into one or more packages, each being handled by a different delivery instance (annotation $+$). Conversely, each delivery instance loads on a delivery truck packages from several $(*)$ order instances and from several $(*)$ previous delivery instances. The packages are delivered one by one: in case of success a single notification (1) is sent to the order instance; in case of failure the single package (1) is either handed over to the order instance or a follow-up delivery instance. These instances in turn collect all incoming notifications or packages $(*)$ before proceeding.

Figure 2 shows an execution of the process as a *partially ordered run* [6]. The execution involves two instances of order and two instances of delivery; order_1 is split into two packages, one handled by $\mathsf{delivery}_1$ and one by $\mathsf{delivery}_2$; the only package of order_2 cannot be delivered in the first attempt and hence is rescheduled to participate in $\mathsf{delivery}_2$ that also handles the second package of order_1. Behavior of this kind naturally arises in an artifact-centric setting and cannot be expressed with service choreographies.

3 Interaction in Artifact Choreographies

When relating the partially ordered run of Fig. 2 to the artifact choreography of Fig. 1 we see that the run satisfies all requirements of the choreography model. Yet, the run also exhibits crucial properties that are not reflected in the model.

(1) The choreography allows a variant of the run of Fig. 2 where the undelivered second package of order_1 is just dropped and not handed over to order_1. In another variant $\mathsf{delivery}_2$ could send 6 messages to order_2 instead of 1. Both kinds of runs are intuitively undesired and should be excluded. Intuitively, not only each artifact instance has a life-cycle to complete, but also *each artifact interaction has a life-cycle* to complete; such a life-cycle is not specified in the choreography.

(2) The choreography would also allow for a run that hands the undelivered package of $\mathsf{delivery}_2$ over to order_2 instead of its original order_1. Although the interaction completes, it completes with the wrong participants. Likewise, one can easily think of a process where such a forwarding of packages to another artifact instance is required. Currently, the choreography does not specify *which* instances interact with each other, but only *how many*.

In other words, the many-to-many relations between artifact instances require a more detailed artifact model than just expressing cardinality constraints.

In particular, the language has to describe (1) the life-cycle of an interaction between artifact instances, and (2) which instances synchronize with each other.

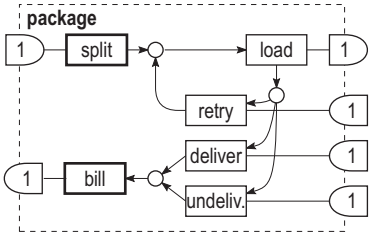

Fig. 4. The artifact package models the life-cycle of an interaction between an instance of order and an instance of delivery (Fig. 1), but not which instances synchronize with each other.

Artifact interaction life-cycle. In the following, we show that the desired interaction between artifact instances can easily be described by the life-cycle of a new, meaningful artifact. Figure 3 decomposes the desired run of Fig. 2 in a specific way. Instead of considering an *asynchronous interaction* between orders and deliveries, Fig. 3 describes exactly the same behavior as Fig. 2 in terms of *synchronous interaction* of orders with packages and packages with deliveries. The dashed lines describe which transitions occur *synchronously*, e.g., split of order_1, $\mathsf{package}_1$, and $\mathsf{package}_3$ occur *synchronously*.

The synchronous interactions caught in the packages describe how we expect the artifact-interactions to complete. These can easily be modeled as a separate artifact package as shown in the proclet of Fig. 4. The package interacts on either side with *exactly one* instance of order and delivery, i.e., it describes the life-cycle of one interaction between two related instances. The choreography of Fig. 1 can be refined to reflect this artifact-interaction life-cycle by placing proclet package between order and delivery. The refinement comes with a paradigm shift: a channel

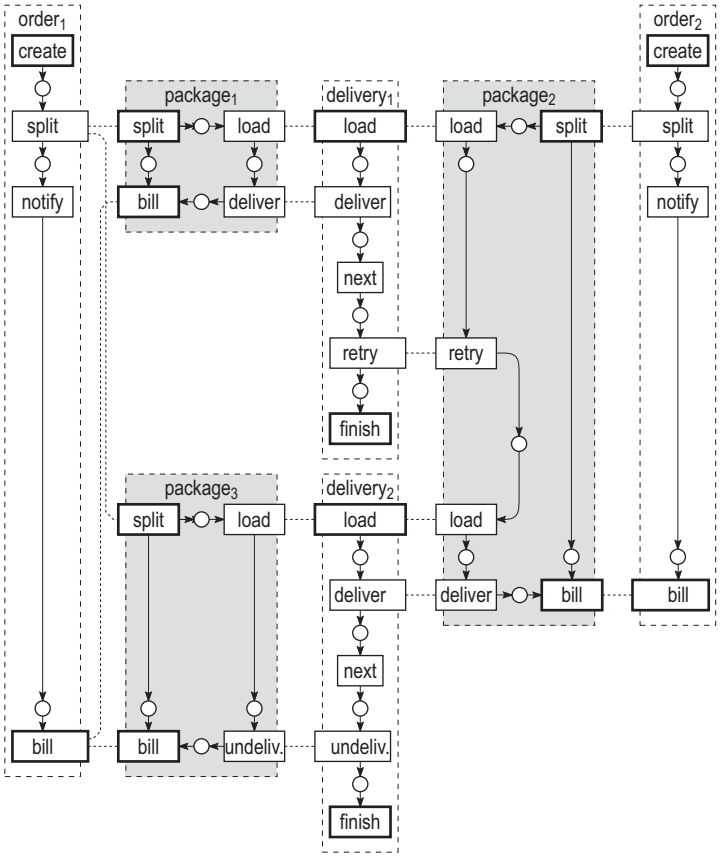

Fig. 3. The messages exchanged between the artifacts in Fig. 2 follow the packages handled in the process.

between two transitions now specifies *synchronous occurrences of transitions* instead of message exchange.

4 Conclusion: Data Specification at the Interaction Level

In this paper, we have shown that artifact choreographies naturally describe behavior that cannot be expressed by services. By lifting the underling data model to the behavioral specification, artifact choreographies particularly express many-to-many relations between artifact instances. Section 3 showed that a *complete* artifact choreography requires to specify *life-cycles of artifact interactions*. To this end, a choreography can be refined with further artifacts.

Two main challenges remain open in this context. First, a choreography description language needs to describe *which* instances interact with each other. It particularly needs to express that an instance A_1 of an artifact A synchronizes with instances $B_1, \ldots, B_k$ of an artifact B which possibly have not been created yet. In our example, a *delivery* is only instantiated *after* all participating *orders*

have been split. A possibility could be to adapt WS-BPEL's *correlation handling* mechanism [7] to the artifact-centric setting.

Second, as artifact interactions can be very complex, it may be reasonable to *synthesize* the artifacts that describe artifact interaction life-cycles. An approach from controller synthesis allows to automatically complete a given choreography in case of 1-to-1 relations [4]. It is worth exploring whether the approach can be leveraged to many-to-many relations.

Alternatively, *process mining* techniques [8] might be applied in this context. Process mining comprises techniques to discover process models from observed behaviors. Such behaviors are extracted from the execution logs of running systems. For an artifact-centric setting, the recorded executions would contain events of artifact-life cycles as well as of artifact interactions. So, execution logs could be an alternative source of information to obtain artifact interaction life-cycles which then lead to a complete artifact choreography.

In controller synthesis, as well as in process mining, the open problem is concerned with the fact that these techniques assume service instances to work in isolation w.r.t. other instances for the same service. In this paper, we have shown that artifact choreographies introduce many-to-many relations among artifacts, which do not exist in traditional service-oriented approaches. As a consequence, the definition of a case concept needs to be rethought. For instance, coming back to the working example, there is no evident preference to consider a diverse case for each order, rather than for each delivery. Every order is associated to several delivery, but also every delivery is associated to several orders.

Acknowledgements. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 257593 (ACSI).

References

1. Nigam, A., Caswell, N.: Business artifacts: An approach to operational specification. IBM Systems Journal **42** (2003) 428–445
2. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. **32** (2009) 3–9
3. Fritz, C., Hull, R., Su, J.: Automatic construction of simple artifact-based business processes. In: ICDT'09. Volume 361 of ACM ICPS. (2009) 225–238
4. Lohmann, N., Wolf, K.: Artifact-centric choreographies. In: ICSOC 2010. Volume 6470 of LNCS., Springer (2010) 32–46
5. van der Aalst, W., Barthelmess, P., Ellis, C., Wainer, J.: Proclets: A Framework for Lightweight Interacting Workflow Processes. Int. J. Cooperative Inf. Syst. **10** (2001) 443–481
6. Engelfriet, J.: Branching processes of Petri nets. Acta Informatica **28** (1991) 575–591
7. Web Services Business Process Execution Language Version 2.0, 11 April 2007. OASIS Standard (2007)
8. van der Aalst, W., Reijers, H., Weijters, A., van Dongen, B., Medeiros, A., Song, M., Verbeek, H.: Business Process Mining: An Industrial Application. Information Systems **32** (2007) 713–732