Checking Behavioral Conformance of Artifacts

Dirk Fahland Massimiliano de Leoni
Boudewijn F. van Dongen
Wil M.P. van der Aalst,
Eindhoven University of Technology, The Netherlands
(d.fahland|m.d.leoni|b.f.v.dongen|w.m.p.v.d.aalst)@tue.nl

March 24, 2011

The usefulness of process models (e.g., for analysis, improvement, or execution) strongly depends on their ability to describe reality. *Conformance checking* is a technique to validate how good a given process model describes recorded executions of the actual process. Recently, *artifacts* have been proposed as a paradigm to capture dynamic, and inter-organizational processes in a more natural way. *Artifact-centric processes* drop several restrictions and assumptions of classical processes, e.g., process instances cannot be considered in isolation as instances in artifact-centric processes may overlap and interact with each other. This significantly complicates conformance checking; the entanglement of different instances complicates the quantification and diagnosis of misalignments. This paper is the first paper to address this problem. We show how conformance checking of artifact-centric processes can be decomposed into a set of smaller problems that can be analyzed using conventional techniques.

Keywords: artifacts, process models, conformance, overlapping process instances

Contents

1 Introduction **5**

2 Artifacts **8**

2.1 Artifacts: An Example 8
2.2 Recording Artifact Behavior: Instance-Aware Logs 9
2.3 Proclets 11
2.3.1 Syntax of Proclets 11
2.3.2 Semantics of Proclets by an Example 13
2.3.3 Formal Definitions - Semantics 14

3 The Artifact Conformance Problem **19**

4 Solving Interaction Conformance **20**

4.1 Reducing Artifact Conformance to Existing Techniques 20
4.2 Structural Viewpoint: a Proclet and its Environment 20
4.3 Behavioral Viewpoint: Extending Cases to Interaction Cases 21
4.4 The Decomposition is Correct 22

5 Conformance Checking Techniques for Artifacts **26**

5.1 Mapping from Proclets to Reset Nets 26
5.2 Mapping Interaction Cases to Cases of Inhibitor-Reset Nets 31
5.3 Replay Techniques for Inhibitor-Reset Nets 32

6 Operationalization in ProM **34**

7 Related Work **36**

8 Conclusion **37**

1 Introduction

Business process models have become an integral part of modern information systems where they are used to document, execute, monitor, and optimize business processes. However, many studies show that models often deviate from reality (see. [13]). To avoid building on quicksand it is vital to know in advance to what extent the model conforms to reality.

Conformance checking is the problem of determining how good a given process model M describes process executions that can be observed in a running system S in reality. Several conformance metrics and techniques are available [15, 9, 19, 11, 14, 3]. The most basic metric is *fitness* telling whether M can *replay* every observed execution of S. In case M cannot replay some (or all) of these executions, the model M needs to be changed to mach the reality recorded by S (or the systems and/or its underlying processes are changed to align both).

Existing conformance checking techniques assume rather simple models where process instances can be considered in isolation. However, when looking at the data models of ERP products such as SAP Business Suite, Microsoft Dynamics AX, Oracle E-Business Suite, Exact Globe, Infor ERP, and Oracle JD Edwards EnterpriseOne, one can easily see that this assumption is *not* valid for real-life processes. There are one-to-many and many-to-many relationships between data *objects*, such as customers, orderlines, orders, deliveries, payments, etc. For example, an online shop may split its customers' *quotes* into several *orders*, one per supplier of the quoted items, s.t. each order contains items *for several customers.* Consequently, several customer cases synchronize on the same order at a supplier, and several supplier cases synchronize on the same quote of a customer. In consequence, we will not be able to identify a unique notion of a *process instance* by which we can trace and isolate executions of such a process, and classical modeling languages are no longer applicable [16, 12, 5].

The fabric of real-life processes cannot be straightjacketed into monolithic processes. Therefore, we need to address two problems:

(1) Find a modeling language $\mathcal{L}$ to express process executions where several cases of different objects overlap and synchronize.

(2) Determine whether a process model M expressed in $\mathcal{L}$ adequately describes actual executions of a dynamic and inter-organizational processes in reality — despite the absence of process instances.

The first problem is well-known [16, 12, 5] and several modeling languages have been proposed to solve it culminating in the stream of *artifact-centric process modeling* that emerged in recent years [16, 12, 5, 7, 10]. In short, an *artifact instance* is an object that participates in the process. It is equipped with a life-cycle that describes the states and possible transitions of the object. An *artifact* describes a class of similar objects, e.g., all *orders*. A process model then describes how artifacts interact with each other, e.g., by exchanging messages [7, 10]. Note that several instances of one artifact may interact with several instances of another artifact, e.g., when placing two orders consisting of multiple items with an electronic bookstore items from both orders may end up in the same delivery while items in the same order may be split over multiple deliveries.

In this paper we use *proclets* [16] as a modeling language for artifacts to study and solve the second problem. A proclet describes one artifact, i.e., a class of objects with their own life cycle, together with the interface to the other proclets. A *proclet system* connects the interfaces of its proclets via unidirectional channels thus allowing the life-cycles of the instances of the connected proclets to interact with each other by exchanging messages; one instance may send a message to multiple other instances, or an instance may receive messages from multiple instances.

After selecting proclets as a representation, we can focus on the second problem; determine whether a given proclet system $\mathcal{P}$ allows for the behavior recorded by the actual information system S, and if not, to which degree $\mathcal{P}$ deviates from S and where. The problem is difficult because S does not structure its executions into isolated process instances. For this reason we develop the notion of an *instance-aware log.* The system S records executed life-cycle cases of its objects in separate logs $L_1, \ldots, L_n$ — one log per class of objects. Each log consists of several cases, and each event in a case is associated to a specific object and also stores with which other objects (having a case in another log) the event interacted (by sending or receiving messages). The *artifact conformance problem* then reads as follows: given a proclet system $\mathcal{P}$ and instance-aware logs $L_1, \ldots, L_n$, can the proclets of $\mathcal{P}$ be instantiated s.t. the life-cycles of all proclets and their interactions "replay" $L_1, \ldots, L_n$?

Depending on how objects in S interact and overlap, a single execution of S can be very long, possibly spanning the entire lifetime of S which results in having to *replay all cases of all logs at once.* Depending on the number of objects and cases, this may turn out infeasible for conformance checking with existing techniques.

Proclets may also be intertwined in various ways. This makes conformance checking a computationally challenging problem. Analysis becomes intractable when actual instance identifiers are taken into account. Existing techniques simply abstract from the identities of instances and their interactions. Therefore, we have developed an approach to decompose the problem into a set of smaller problems: we minimally enrich each case in each log to an *interaction case*, describing how one object evolves through the process *and* synchronizes with other other objects, according to other cases in the other logs. We then show how to abstract a given proclet system $\mathcal{P}$ for each proclet P to an abstract proclet system $\mathcal{P}|_P$ s.t. $\mathcal{P}$ can replay $L_1, \ldots, L_n$ iff for each proclet P of $\mathcal{P}$ the abstract proclet system $\mathcal{P}|_P$ can replay each interaction case of P. As an interaction case focuses on a single instance at a time (while taking its interactions into account), existing conformance checkers [14, 3, 2] can be used to check conformance.

The remainder of this report is structured as follows. In Section 2, we recall how artifacts allow describing processes where cases of different objects overlap and interact.

There, we also introduce the notion of an *instance-aware event log* that contains just enough information to reconstruct executions of such processes. Further, proclets are introduced as a formal language to describe such processes. Section 3 then formally states the conformance checking problem in this setting, and Section 4 presents our technique of decomposing proclet systems and logs for conformance checking. Section 5 provides details how to reduce the decomposed conformance checking problem to a conformance problem on Petri nets. The entire approach is implemented in the process mining toolkit ProM; Section 6 presents the tool's functionality and shows how it can discover deviations in artifact-centric processes. Related work is presented in Section 7. Section 8 concludes the paper.

2 Artifacts

This section recalls how the artifact-centric approach allows to describe processes where cases of different objects overlap and interact. We present proclets as a formal model to describe such processes. Moreover, we introduce the new notion of an instance-aware log which is a minimal extension of a classical event log to record executions of processes with overlapping instances.

2.1 Artifacts: An Example

To motivate all relevant concepts and to establish our terminology, we consider a backend process of a CD online shop that serves as a running example in this paper. The CD online shop offers a large collection of CDs from different suppliers to its customers. Its backend process is triggered by a customer's request for CDs and the shop returns a *quote* regarding the offered CDs. If the customer accepts, the shop splits the quote into *several orders*, one at each CD supplier. One order handles *all quoted CDs* by the same supplier. The order then is executed and the suppliers ship the CDs to the shop which distributes CDs from different orders according to the original quotes. Some CDs may be unavailable at the supplier; in this case notifications are sent to the CD shop which forwards the information to the customer. The order closes when all CDs are shipped and all notifications are sent. The quote closes after the customer rejected the quote, or after notifications, CDs, and invoice have been sent.

Figure 2.1 models the above process in terms of a *proclet system* consisting of two *proclets*: one describing the life-cycle of *quotes* and the other describing the life-cycle of *orders*. Note that Fig. 2.1 abstracts from interactions between the CD shop and the customers. Instead the focus is on interactions between quotes in the CD shop and orders handled by suppliers.

The distinctive quality in the interactions between quotes and orders is their *cardinality*: each quote interacts with *several* orders, and each order interacts with *several* quotes. That is, we observe *many-to-many* relations between quotes and orders. For example, consider a process execution involving two quote *instances*: one over $\mathsf{CD_a}$ (q_1) and the other over $\mathsf{CD_a}$, $\mathsf{CD_b}$, and $\mathsf{CD_c}$ (q_2). $\mathsf{CD_b}$ and $\mathsf{CD_c}$ have the same supplier, $\mathsf{CD_a}$ has a different supplier. Hence, the quotes are split into two order instances (o_1 and o_2). In the execution, $\mathsf{CD_a}$ and $\mathsf{CD_b}$ turn out to be available whereas $\mathsf{CD_c}$ is not. Consequently, $\mathsf{CD_a}$ is shipped to the first quote, and $\mathsf{CD_a}$ and $\mathsf{CD_b}$ are delivered to the second quote. The second quote is also notified regarding the unavailability of $\mathsf{CD_c}$. This

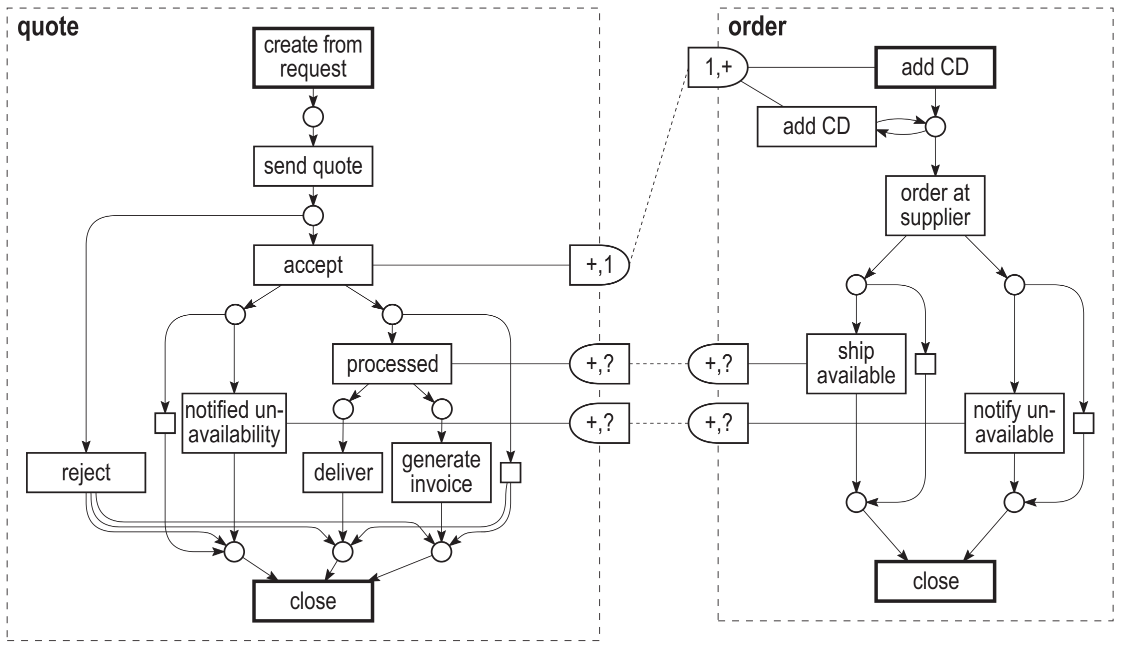


Figure 2.1: A proclet system describing the back-end process of a CD online shop. A customer's quote is split into several orders according to the suppliers of the CDs; an order at a supplier handles several quotes from different customers.

execution gives rise to the following cases of quote and order:

q_1 : create, send, accept, processed, deliver, generate, close
q_2 : create, send, accept, notified, processed, deliver, generate, close
o_1 : add CD, add CD, order, ship, close
o_2 : add CD, add CD, order, notify, ship, close

These cases interact with each other as illustrated in Fig. 2.2.

2.2 Recording Artifact Behavior: Instance-Aware Logs

The task of checking whether a given process model accurately describes the processes executed in a running system S requires that S records the relevant events in a *log*. Classically, each process execution in S corresponds to a *case* running in isolation. Such a process instance can be represented by the sequence of events that occurred. In an artifact-centric process like Fig. 2.1, one cannot abstract from interactions and many-to-many relations; quotes and orders are interacting in a way that cannot be abstracted away.

Relating events of different cases to each other is known as *event correlation*; see [4] for a survey of correlation patterns. A *set of events* (of different cases) is said to be correlated by some property P if each event has this property P. The set of all correlated events

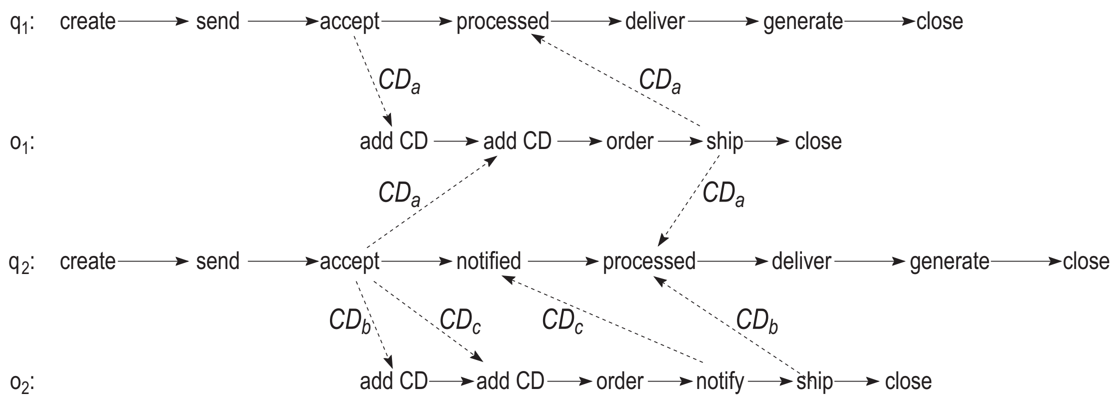


Figure 2.2: An execution of the CD shop process involving two quote cases and two order cases that interact with each other in a many-to-many fashion.

defines a *conversation*. For instance in Fig. 2.2, events accept and processed of q_1, and events add CD and ship of o_1 form a conversation. Various correlation mechanisms to define and set the correlation property of an event are possible [4]. In this paper, we do not focus on the actual correlation mechanism. We simply assume that such correlations have been derived; these are the connections between the different instances in Fig. 2.2.

To abstract from a specific correlation mechanism regarding events in a log, we introduce the notion of an *instance-aware log*. Let e be an event. Event correlation is used to derive the *sender instances*, i.e., the instances that sent the messages which e consumed, as well as the *recipient instances*, i.e., the instances that receive the messages which the occurrence of e has produced. Note that we assume asynchronous interaction between different instances. Since multiple messages can be sent/received to/from the same instance, correlation data are stored as *multisets* of instance ids. A multiset $m \in \mathbb{N}^{\mathcal{I}}$ over a set $\mathcal{I}$ is technically a mapping $m : \mathcal{I} \rightarrow \mathbb{N}$ defining how often each $id \in \mathcal{I}$ occurs in m; $[]$ denotes the empty multiset.

Definition 1 (Instance-aware events)**.** Let $\Sigma = \{a_1, a_2, \ldots, a_n\}$ be a finite set of *event types*, and let $\mathcal{I} = \{id_1, id_2, \ldots\}$ be a set of *instance identifiers*. An *instance-aware event* e is a 4-tuple $e = (a, id, SID, RID)$ where $a \in \Sigma$ is the event type, id is the instance for which e occurred, $SID = [sid_1, \ldots, sid_k] \in \mathbb{N}^{\mathcal{I}}$ defines the multiset of sender instances, and $RID = [rid_1, \ldots, rid_l] \in \mathbb{N}^{\mathcal{I}}$ defines the multiset of recipient instances. Let $\mathcal{E}(\Sigma, \mathcal{I})$ denote the set of all instance-aware events over Σ and $\mathcal{I}$.

Consider for example the third event of q_2 in Fig. 2.2. This instance aware event is denoted as $(\mathsf{accept}, q_2, , [], [o_1, o_2, o_2])$. The fifth event of q_2 is denoted as $(\mathsf{processed}, q_2, [o_1, o_2], [])$.

All events of one instance of an object o define a case; all cases of o define the log of o. An execution of the entire process records the cases of the involved object instances in different logs that together constitute an instance-aware log.

Definition 2 (Instance-aware cases and logs)**.** An *instance-aware case* $\sigma = \langle e_1, \ldots, e_r \rangle \in \mathcal{E}(\Sigma, \mathcal{I})^*$ is a finite sequence of instance-aware events occurring all in the same instance $id \in \mathcal{I}$. Let $L_1, \ldots, L_n$ be sets of finitely many instance-aware cases s.t. no two cases use the same instance id. Further, let $<$ be a total order on all events in all cases s.t. $e < e'$ whenever e occurs before e' in the same case.[1] Then $\mathcal{L} = (\{L_1, \ldots, L_n\}, <)$ is called an *instance-aware log.*

For example, the instance-aware cases of Fig. 2.2 are the following:

$$\begin{array}{rl}
\sigma_{q1}: & \langle(\mathsf{create}, q_1, [], []), (\mathsf{send}, q_1, [], []), (\mathsf{accept}, q_1, [], [o_1]), (\mathsf{processed}, q_1, [o_1], []), \\
& (\mathsf{deliver}, q_1, [], []), (\mathsf{generate}, q_1, [], []), (\mathsf{close}, q_1, [], [])\rangle \\
\sigma_{q2}: & \langle(\mathsf{create}, q_2, [], []), (\mathsf{send}, q_2, [], []), (\mathsf{accept}, q_2, [], [o_1, o_2, o_2]), (\mathsf{notified}, q_2, [o_2], []), \\
& (\mathsf{processed}, q_2, [o_1, o_2], []), (\mathsf{deliver}, q_2, [], []), (\mathsf{generate}, q_2, [], []), (\mathsf{close}, q_2, [], [])\rangle \\
\sigma_{o1}: & \langle(\mathsf{add\ CD}, o_1, [q_1], []), (\mathsf{add\ CD}, o_1, [q_2], []), (\mathsf{order}, o_1, [], []), (\mathsf{ship}, o_1, [], [q_1, q_2]), \\
& (\mathsf{close}, o_1, [], [])\rangle \\
\sigma_{o2}: & \langle(\mathsf{add\ CD}, o_2, [q_2], []), (\mathsf{add\ CD}, o_2, [q_2], []), (\mathsf{order}, o_2, [], []), (\mathsf{notify}, o_2, [], [q_2]), \\
& (\mathsf{ship}, o_2, [], [q_2]), (\mathsf{close}, o_1, [], [])\rangle
\end{array}$$

Together these instances form an instance-aware log with an ordering relation $<$, e.g., $(\mathsf{accept}, q_1, [], [o_1]) < (\mathsf{add\ CD}, o_1, [q_1], [])$.

A process that follows from interactions of objects $o_1, \ldots, o_n$ records each case of object o_i in the respective log L_i. The essential correlation information of an event is represented by each event's instance id and the sets of sender and receiver ids. Note that either set is empty if the event is not receiving or not sending any message. The order $<$ puts events of different cases into an order, and is typically defined by the timestamps of the event occurrences.

2.3 Proclets

In the recent years, the *artifact-centric approach* emerged as a paradigm to describe processes like in our CD shop example where several cases of one object interact with several cases of another object. Different languages for describing artifacts have been proposed [16, 12, 5, 7, 10]. In the following, we use *proclets* [16] to study instantiation of artifacts and the many-to-many interactions between different artifact instances in a light-weight formal model. A proclet models an artifact life-cycle as a labeled Petri net where some transitions are attached to *ports.* A *proclet system* consists of a set of proclets together with *channels* between the proclets' ports. Annotations at the ports lift the cardinality constraints of the underlying data model to the life-cycle model, i.e., they specify how many instances interact with each other via a channel.

We first introduce the formal syntax of proclets, then informally explain their semantics along our running example, and then give a full formal definition of their semantics.

2.3.1 Syntax of Proclets

Proclets are based on labeled Petri nets.

[1]Note that technically two different events could have the same properties (e.g., in a loop). We assume these to be different, but did not introduce additional identifiers.

Definition 3 (Petri net, labeled)**.** A *Petri net* $N = (S, T, F, \ell)$ consists of a set S of *places*, a set T of *transitions* disjoint from S, arcs $F \subseteq (S \times T) \cup (T \times S)$, and a *labeling* $\ell : T \to \Sigma \cup \{\tau\}$ assigning each transition t an action name $\ell(t) \in \Sigma$ or the invisible label τ.

A proclet is a labeled Petri net where some transitions are attached to a *port* which enables them to communicate with other proclets via message exchange.

Definition 4 (Proclet)**.** A *proclet* $P = (N, \mathit{ports})$ consists of a Petri net $N = (S, T, F, \ell)$ and a set of *ports* $\subseteq 2^T \times \{\mathit{in}, \mathit{out}\} \times \{?, 1, *, +\} \times \{?, 1, *, +\}$ where each port $p = (T^p, \mathit{dir}_p, \mathit{card}_p, \mathit{mult}_p)$

1. is associated to a set $T^p \subseteq T$ of transitions s.t. for all $t_1, t_2 \in T_p$ holds: $\ell(t_1) = \ell(t_2) \neq \tau$;
2. has a direction of communication (in_p: incoming port, the associated transitions receive a message, out_p: outgoing port, the associated transitions send a message);
3. has a *cardinality* $\mathit{card}_p \in \{?, 1, *, +\}$ specifying how many messages may or have to be sent or received upon an occurrence of one $t \in T^p$;
4. has a *multiplicity* $\mathit{mult}_p \in \{?, 1, *, +\}$ specifying how often all transitions T^p may occur together during the lifetime of an instance of proclet P; and
5. and no two ports share a transition: $T^p \cap T^q = \emptyset$, for all $p, q \in \mathit{ports}, p \neq q$.

Figure 2.1 shows two proclets. Each has three ports. The output port of accept has cardinality + (one event may send messages to multiple orders) and multiplicity 1 (this is done only once per quote). The input port of add CD has a cardinality of 1 (each individual input message triggers one of the add CD transitions) and a multiplicity + (at least one message is received during the life-cycle of an order).

We generally assume that each proclet P has a unique transition $\mathit{create}(P)$ with an empty pre-set (no incoming arcs), and a unique transition $\mathit{final}(P)$ with an empty post-set. These transitions denote actions to create and finish an instance of P respectively. We also write $p = \mathit{port}(t)$ iff the port $p = (T_p, \mathit{dir}, \mathit{card}, \mathit{mult})$ is associated to transition $t \in T_p$.

Introducing P implicitly introduces its components $N_p = (S_P, T_P, F_P, \ell_P)$ and port_p; the same applies to P',P_1, etc. and their components $N' = (S', T', F', \ell')$ and port', and $N_1 = (S_1, T_1, F_1, \ell_1)$ and port_1, respectively. This notation also applies to other structures later on.

Definition 5 (Proclet system)**.** A *proclet system* $(\{P_1, \ldots, P_n\}, C)$ 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 $p, q \in \bigcup_{i=1}^{n} \mathit{ports}_i$ with direction of p being *in* and direction of q being *out*.

Our notion of a proclet system assumes that each output port p is related to exactly one input port q (and vice versa) that together constitute a channel (p, q). We will also write $P_1 \oplus \ldots \oplus P_n$ as a short hand for the proclet system $(\{P_1, \ldots, P_n\}, C)$.

2.3.2 Semantics of Proclets by an Example

In the following, we explain the semantics of proclets by the running example of the CD shop introduced in Section 2.1. We specifically focus on how proclets are instantiated, and how ports and channels express many-to-many relations between instances. The full formal semantics of proclets are given in Section 2.3.3.

Two proclets Figure 2.1 shows the proclet system that describes our online shop. The left proclet describes the life-cycle of the quote, the right proclet describes the life-cycle of the order. A process execution emerges by instantiating proclets and letting these instances exchange messages asynchronously via the three channels. Each channel has one *input port* and one *output port*; the direction of the channel is indicated by the port's bow.

Proclet for Quote A new instance of proclet quote is created whenever a customer requests a new quote from the CD shop. The proclet model abstracts from the interaction with the customer and describes the corresponding interaction as actions that are internal to the quote. After the quote has been sent to the customer, the customer will either reject or accept the quote.

When the quote is rejected, action close is enabled which will terminate this instance of a quote. In this case, no communication with another proclet instance occurred.

Communication to other proclet instances If the quote is accepted, messages are placed at the input port of the top-most channel to proclet order. The input port specification $(+, 1)$ describes that one occurrence of the action accept places one or more messages (cardinality $+$) at the output port of the channel; the messages may be addressed to *different or the same* instances of the order proclet. This way, the proclet describes that one quote (consisting of many CD items) may be split into several orders depending on the available suppliers as described in Sect. 2.1. In addition, this output port may be only used once during the lifetime of the proclet instance as indicated by mutliplicity 1. This constraint captures that each quote is processed only once and no retries will be made. After action accept occurred, the quote proclet has to wait for messages on the other channels to proceed.

The messages placed on the top-most channel by the accept message are sent to different instances of the order proclet. Each instance of an order proclet (created independently for each CD supplier as described in Sect. 2.1) collects requests for CDs from different quotes. This is captured by the port specification $(1, +)$ as follows: the action add CD will receive one request message for a CD from one quote at a time and include it in the order (cardinality 1). However, one or more messages (coming from different quotes) can be received via the input port during the lifetime of the order proclet (multiplicity $+$).

Thus, the multiplicities between instances of the artifacts quote and order are reflected by the possibilities their proclet instances can communicate with each other via connecting channels. Specifically, one quote sends to multiple orders, and one order receives from multiple quotes.

Proclet for Order After all quotes have been added to the order and the order is closed, the order is sent to the corresponding supplier (action order at supplier) who then ships available CDs and sends the corresponding invoice to the CD shop, and by sending notifications about undeliverable CDs to the CD shop. Notifying about undeliverable CDs may be skipped if all CDs can be delivered which is expressed by the internal transition next to notify available. Correspondingly, shipping CDs and generating invoices may be skipped together if all CDs are undeliverable. The instance of the order proclet terminates after these actions occurred.

Many-to-Many Communication to Complete an Order When shipping CDs (occurrence of action ship available), proclet order generates multiple messages (cardinality + at the output port) representing multiple deliveries that are split according to the different quotes added earlier. Sending CDs may occur at most once (multiplicity ?) during the lifetime of the order proclet because sending may be skipped (if all CDs are unavailable) but will not be retried. Correspondingly, multiple notifications about undeliverabiliy are sent at most once to the respective orders.

At the opposite end of the channels, the corresponding instances of the quote proclet wait for all shipments or notifications from the different orders to arrive (cardinality +). Either input port might not be used at all (multiplicity ?) in case no or all CDs can be shipped. Depending on the received messages from its orders, the quote proclet notifies the customer about undeliverability (left branch after the accept action) and/or delivers the quote to the customer and generates and sends the corresponding invoice (right branch). Either branch can be skipped by an internal transition depending on the outcome of the orders. The instance of the quote proclet terminates via action close after these actions occurred.

How proclets describe relation between objects According to the specification of the input and output ports of the channels, a quote is split into multiple orders; each order, in turn, handles the shipment of CDs of multiple quotes. This way, the many-to-many relations of the process manifest in the proclet model.

2.3.3 Formal Definitions - Semantics

After this informal introduction to proclet semantics, we now give the corresponding formal definitions. We tailor the semantics for the setting of conformance checking and provide so called *replay semantics*. In a replay semantics, a step of the system model is not fully determined by the state of the model, but may also be based on external input. In our case, this external input will be a recorded system execution, that is, an instance aware log (see Sect. 2.2), which describes which proclet instances exchanged messages with each other.

The replay semantics describes when a sequence σ of instance-aware events *satisfies* all behavioral restrictions stated in a proclet system $P_1 \oplus \ldots \oplus P_n$, i.e., whether an actual execution of the real process follows the life-cycles and relations specified in $P_1 \oplus \ldots \oplus P_n$.

Intuitively, the semantics of proclets follows that of high-level Petri nets: a *configuration* of the system puts tokens on the proclets' places. To distinguish the different instances from each other each token is colored by an *instance id.* The state of a channel is defined by the messages in the channel. Each message contains the sender's instance id_s and the recipient's instance id_r to properly identify which proclet instances are interacting with each other; thus a message is formally a pair (id_s, id_r).

Definition 6 (Configuration of a proclet system)**.** Let $\mathcal{P} = (\{P_1, \ldots, P_n\}, C)$ be a proclet system s.t. $S_{P_i} \cap S_{P_j} = \emptyset$, for all $1 \leq i < j \leq n$. Let $\mathcal{I}$ be a set of *instance ids.* A *configuration* $K = (\mathcal{I}, m_S, m_C)$ of $(\{P_1, \ldots, P_n\}, C)$ is defined as follows:

1. The set $\mathcal{I}$ defines the *active instances.* For each instance $id \in \mathcal{I}$ let $type(id) = P_i$ denote proclet of which id is an instance.
2. The *place marking* m_S defines for each place $s \in S := \bigcup_{i=1}^{n} S_i$ the number of tokens that are on place s in instance id. Formally, $m_S : S \to \mathbb{N}^{\mathcal{I}}$ assigns each place $s \in S$ a multiset of instance ids, i.e., $m_S(s)(id)$ defines the number of tokens on s in id.
3. The *channel marking* m_C defines for each channel $c \in C$ the messages in this channel. Formally $m_C : C \to \mathbb{N}^{\mathcal{I} \times \mathcal{I}}$ is a multiset of pairs of instances ids, i.e., $m_C(c)(id_s, id_r)$ defines the number of messages that are in transit from id_s to id_r in channel c.[2]

The *initial configuration* is $K_0 = (\emptyset, m_{S,0}, m_{C,0})$ with $m_{S,0}(s) = \emptyset$ for all places s, and $m_{C,0}(p, q) = \emptyset$ for all channels (p, q).

A configuration generalizes the notion of a marking of a (case-sensitive) Petri net. The steps of a proclet system generalize the steps of a (case-sensitive) Petri net in the same vein. Each transition t occurs in a specific proclet instance id. For this, t has to be enabled, i.e., all pre-places of t have a token colored id. An occurrence of t consumes one id-colored token from each pre-place and produces an id-colored token on each post-place.

The actual contribution of proclets comes from their ports via which t receives or sends messages. As messages are sent between specific proclet instances, an occurrence of t also has to determine to which instances it sends a message or from which instances it receives a message. Technically, we specify a multiset *SID* of *sender instances* from which t expects to receive a message, and a s multiset *RID* of *recipient instances*, to which t is going to send a message. These sets have to satisfy the constraints of the port p to which t is attached.

[2]We do not consider here the data perspective of the messages. So, we ignore the informative content of messages (i.e., the data fields) and only focus on identifiers of the senders and receivers. The use of data perspective for conformance checking is part of our future work.

A transition t can only occur if it is *enabled* at the given configuration $K = (\mathcal{I}, m_S, m_C)$ in instance id. The enabling of t depends on the *validity* of the multiset SID of sender instances, from which t expects to receive messages.

Definition 7 (Validity of a multiset of senders w.r.t. a transition)**.** Let $P = (N, ports)$ be a proclet and $t \in T_P$ a transition of P. Let SID be a multiset of sender instances, from which t expects to receive messages. SID is *valid* w.r.t. t in proclet instance id at configuration $K = (\mathcal{I}, m_S, m_C)$ iff

- if t is not attached to an input port, then $SID = []$, and
- if t is attached to an input port $p = (T^p, dir_p, card_p, mult_p)$, $t \in T^p$, $dir_p = in$ at channel $c = (q, p) \in C$, then
  1. $card_p = 1$ implies $SID = [id_s]$ and $m_C(c)(id_s, id) > 0$,
  2. $card_p = ?$ implies if $|[(id_s, id) | (id_s, id) \in m_C(c)]| = 0$ then $SID = []$, else $SID = [id_s]$ for some $(id_s, id) \in m_C(c)$,
  3. $card_p \in \{+, *\}$ implies $SID(id_s) = |[(id_s, id) | (id_s, id) \in m_C(c)]|$, for all id_s, and if $card_p = +$, then additionally $SID \neq []$.

Corresponding to receiving messages from expected senders, t also sends message to chosen recipients RID if t is attached to an output port p. The multiset RID also has to satisfy the constraints of p, though the choice of RID does not depend on the contents of the channel as the corresponding message will be produced by t.

Definition 8 (Validity of a multiset of recipients w.r.t. a transition)**.** Let $P = (N, ports)$ be a proclet and $t \in T_P$ a transition of P. Let RID be a multiset of recipient instances, to which t wants to send messages. RID is *valid* w.r.t. t in proclet instance id at configuration $K = (\mathcal{I}, m_S, m_C)$ iff

- if t is not attached to an output port, then $RID = []$, and
- if t is attached to an output port $p = (T^p, dir_p, card_p, mult_p)$, $t \in T^p$, $dir_p = out$, then
  1. $card_p = 1$ implies $|RID| = 1$,
  2. $card_p = ?$ implies $|RID| \in \{0, 1\}$, and
  3. $card_p = +$ implies $|RID| \geq 1$.

If t is not attached to an input and/or output port, then SID and/or RID are, respectively, always empty for each occurrence of transition t.

Definition 9 (Enabled transition)**.** Let $\mathcal{P} = (\{P_1, \ldots, P_n\}, C)$ be a proclet system. Let $K = (\mathcal{I}, m_S, m_C)$ be a configuration. Let t be a transition of a proclet $P_i, i \in 1, \ldots, n$, let id be an instance id of P_i, and let SID and RID multisets of expected senders and intended recipients, respectively.

Transition t is *enabled in instance id* w.r.t. SID and RID at configuration K, written $K \xrightarrow{t, id, SID, RID}$ iff

1. instance *id* has a token on every input place s of t, i.e., for each $s \in {}^\bullet t$ holds: $m_S(s)(id) > 0$,

2. *SID* is a valid set of sender ids w.r.t. t, and

3. *RID* is a valid set of recipient ids w.r.t. t.

Like in Petri nets, an enabled transition t can *occur* which results in a successor configuration K' by consuming tokens from the pre-places of t and producing tokens on the post-places of t. In addition, if t is attached to a port p, then t will also consume messages from p or produce messages on p as specified in *SID* and *RID*.

Definition 10 (Replay semantics)**.** Let $\mathcal{P} = (\{P_1, \ldots, P_n\}, C)$ be a proclet system. Let $K = (\mathcal{I}, m_S, m_C)$ be a configuration. Let t be a transition of a proclet $P_i, i \in 1, \ldots, n$, let *id* be an instance id of P_i, and let *SID* and *RID* be (possibly empty) multisets sender and recipient ids so that $K \xrightarrow{t,id,SID,RID}$ holds (i.e., t is enabled).

An *occurrence* of t in *id* w.r.t. *SID* and *RID* defines the instance-aware event $e = (t, id, SID, RID)$ and the *step* $K \xrightarrow{t,id,SID,RID} K'$ that leads to the *successor configuration* $K' = (\mathcal{I}', m'_S, m'_C)$ as follows:

1. if $t = create(P_i)$, then $\mathcal{I}' = \mathcal{I} \cup \{id\}$, $id \notin \mathcal{I}$, and if $t = final(P_i)$, then $\mathcal{I}' = \mathcal{I} \setminus \{id\}$;

2. the occurrence of t changes the tokens of instance *id* according to Petri net semantics, i.e., for all places $s \in \bigcup_{i=1}^{n} S_{P_i}$ holds: $m'_S(s) = m_S(s) - id$ iff $s \in {}^\bullet t \setminus t^\bullet$, $m'_S(s) = m(s) + id$ iff $s \in t^\bullet \setminus {}^\bullet t$, and $m'_S(s) = m(s)$ otherwise;

3. the occurrence of t changes the messages in the channels attached as required by the port of t, i.e., for each channel $(p, q) \in C$ holds

   a) if $t \in T^q$, i.e., t is attached to input port $q = (T^q, in, card, mult)$, then $m'(p, q) = m(p, q) - [SID(id_s) \cdot (id_s, id) \mid id_s \in SID]$, i.e., *id* receives as many messages $SID(id_s)$ as expected from each $id_s \in SID$; and

   b) if $t \in T^p$, i.e., t is attached to output port $p = (T^p, out, card, mult)$, then $m'(p, q) = m(p, q) + [RID(id_r) \cdot (id, id_r) \mid id_r \in RID]$, i.e., *id* sends as many messages $RID(id_r)$ as intended to each $id_r \in RID$;

   c) $m'(p, q) = m(p, q)$, otherwise.

A sequence $K_0 \xrightarrow{t_1,id_1,SID_1,RID_1} K_1 \xrightarrow{t_2,id_2,SID_2,RID_2} K_3 \xrightarrow{t_3,id_3,SID_3,RID_3} \ldots K_n$ of steps is a *run* iff K_0 is the initial configuration, and for each port $p = (T^p, dir, card, mult)$ holds

1. if $mult = ?$, then there exists at most one $i > 0$ s.t. $t_i \in T^p$,

2. if $mult = 1$, then there exists exactly one $i > 0$ s.t. $t_i \in T^p$,

3. if $mult = +$, then there exists at least one $i > 0$ s.t. $t_i \in T^p$.

Note that input and output ports have slightly asymmetric semantics for cardinalities $+$ and $*$. An input port of transition t requires t consume all messages available in the channel in this case ($+$ requires at least one message to be present for t to be enabled). An output port will let t produce a number of messages, where the number of messages is not fixed. This asymmetry originates in the fact that proclet abstract from the concrete data that would determine the interaction of artifacts, and hence from the specific messages to consume and to produce.

Thus, an execution of the proclet system $\mathcal{P}$ can be observed as a sequence $K_0 \xrightarrow{(t_1,id_1,SID_1,RID_1)} K_1 \xrightarrow{(t_2,id_2,SID_2,RID_2)} \dots K_n$ where each K_{i+1} is the successor configuration of K_i under the instance-aware event $(t_i, id_i, SID_i, SID_i)$.

This semantics also allows to *replay* an instance-aware log $\mathcal{L} = (\{L_1, \dots, L_n\}, <)$ on a given proclet system $\mathcal{P} = P_1 \oplus \dots \oplus P_n$, or to check whether $\mathcal{P}$ can replay $\mathcal{L}$. For this replay, we merge all events of all cases of all logs $L_1, \dots, L_n$ into a single sequence σ of events that are ordered by $<$. $\mathcal{P}$ can replay $\mathcal{L}$ iff the events of σ define an execution of $\mathcal{P}$. For instance, merging the cases $\sigma_{q1}, \sigma_{q2}, \sigma_{o1}, \sigma_{o2}$ of Section 2.2 yields a case that can be replayed in the proclet system of Fig. 2.1.

3 The Artifact Conformance Problem

The problem of determining how accurately a formal process model describes the process implemented in an actual information system S is called *conformance checking problem* [15].

Classically, a system S executes a process as a set of isolated instances. The corresponding *observed* system execution is a sequence of events, called *case*, and a set of cases is a *log* L. The semantics of a formal process model M define the set of valid process executions in terms of sequences of M's actions. Conformance of M to L can be characterized in several dimensions [15]. In the following, we consider only *fitness*. This is the most dominant conformance metric that describes to which degree a model M can replay all cases of a given log L, e.g., [3]. M fits L less, for instance, if M executes some actions in a different order than observed in L, or if L contains actions not described in M. Several conformance checking techniques for process models are available [3, 15, 9, 19, 11, 14]. The more robust techniques, e.g., [3], find for each case $\sigma \in L$ an execution σ' of M that is as *similar* as possible to σ; the similarity of all σ to their respective σ' defines the fitness of M to L.

A proclet system raises a more general conformance checking problem, because a case contains events of several instances of proclets that all may interact with each other. In our example from Section 2, handling one quote of the CD shop involves *several* order instances, i.e., the case spans one quote instance and several order instances. From a different angle, a complete handling of an order involves *several* quote instances.

In the light of this observation, we identify the following *artifact conformance problem.* A system records events in an instance-aware event log $\mathcal{L}$. Each event can be associated to a specific proclet P of a proclet system $\mathcal{P}$, knows the instance in which it occurs and the instances with which it communicates. Can the proclet system $\mathcal{P}$ replay $\mathcal{L}$? If not, to which degree does $\mathcal{P}$ deviate from the behavior recorded in $\mathcal{L}$?

4 Solving Interaction Conformance

A naïve solution of the artifact conformance problem would replay the entire log $\mathcal{L}$ on the proclet system $\mathcal{P}$, by instantiating proclets and exchanging messages between different proclet instances. This approach can become practically infeasible because of the sheer size of $\mathcal{L}$ and the number of active instances. Moreover, existing techniques would be unable to distinguish the difference instances. For this reason, we decompose the problem and reduce it to a classical conformance checking problem.

4.1 Reducing Artifact Conformance to Existing Techniques

A simple decomposition of the artifact conformance problem would be to take from $\mathcal{L} = (\{L_1, \dots, L_n\}, <)$ for each proclet P, and each instance P^{id} of P the case σ^{id} of P^{id} from the respective log. σ^{id} describes how P^{id} evolved according to its life-cycle. However, completing the life-cycle not only depends on events of P^{id} but also on events that sent messages to P^{id} or received messages from P^{id}. So, all events of σ^{id} *together* with all events of $\mathcal{L}$ that exchange messages with P^{id} constitute the *interaction case* $\overline{\sigma^{id}}$ of P^{id}. It contains all behavioral information regarding how P^{id} interacts with other proclet instances.

An interaction case $\overline{\sigma^{id}}$ of a proclet instance P^{id} gives rise to the following conformance problem. The proclet system $\mathcal{P}$ *fits* $\overline{\sigma^{id}}$ iff $\overline{\sigma^{id}}$ (1) follows the life-cycle of P, and (2) has as many communication events as required by the channels in $\mathcal{P}$. The *interaction conformance* problem is to check how good $\mathcal{P}$ fits all interactions cases of all proclets.

We will show in the next section that decomposing artifact conformance into interaction conformance is correct: if $\mathcal{P}$ fits $\mathcal{L}$, then $\mathcal{P}$ fits each interaction case of each proclet P of $\mathcal{P}$; and if $\mathcal{P}$ does not fit $\mathcal{L}$, then there is an interaction case of a proclet $\mathcal{P}$ to which $\mathcal{P}$ does not fit. As each interaction case is significantly smaller than $\mathcal{L}$ and involves only one proclet instance, the conformance checking problem becomes feasible and can be solved with existing techniques.

4.2 Structural Viewpoint: a Proclet and its Environment

Our aim is to decompose the conformance checking problem of a proclet system $\mathcal{P} = P_1 \oplus \dots \oplus P_n$ w.r.t. $\mathcal{L}$ into a set of smaller problems: we check interaction conformance for each proclet P_i. Interaction conformance of P_i considers the behavior of P_i together with the immediate interaction behavior of P_i with all other proclets $P_1, \dots, P_{i-1}, P_{i+1}, \dots, P_n$.

We capture this immediate interaction behavior by abstracting $P_1, \dots, P_{i-1}$, $P_{i+1}, \dots, P_n$ to an environment $\overline{P_i}$ of P_i. $\overline{P_i}$ is a proclet that contains

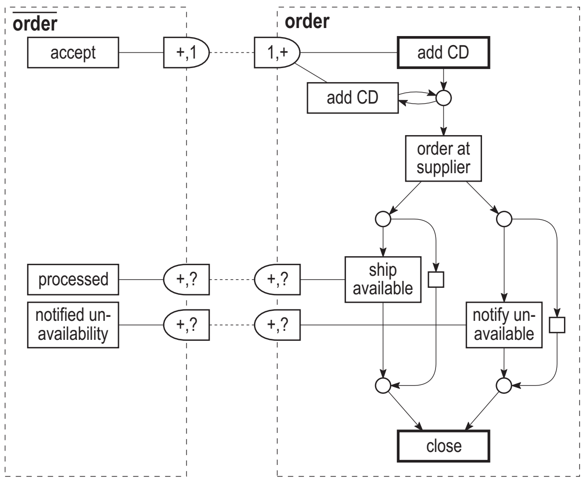


Figure 4.1: The proclet order of Fig. 2.1 together with its environment $\overline{\text{order}}$.

just those transitions of $P_1, \dots, P_{i-1}, P_{i+1}, \dots, P_n$ at the remote ends of the channels that reach P_i — together with the corresponding ports for exchanging messages with P_i. Obviously, occurrences of transitions of $\overline{P_i}$ are unconstrained up to messages sent by P_i. Composing P_i and $\overline{P_i}$ yields the proclet system $P_i \oplus \overline{P_i}$ in which we can replay the interaction cases of P_i.

Figure 4.1 shows the proclet order together with its abstracted environment $\overline{\text{order}}$ from the proclet system of Fig. 2.1.

The formal definition reads as follows.

Definition 11 (Environment Abstraction)**.** Let $\mathcal{P} = (\{P_1, \dots, P_n\}, C)$ be a proclet system, $P_i = (N_i, \mathit{ports}_i), N_i = (S_i, T_i, F_i), i = 1, \dots, n$. We write $t \in T^p$ if a transition t is attached to port p. The channels that reach P_i are $C_i = \{(p, q) \in C \mid (T^p \cup T^q) \cap T_i \neq \emptyset\}$. The transitions at the remote ends of these channels are $\overline{T_i} = \{t \mid (p, q) \in C_i, t \in (T^p \cup T^q) \setminus T_i\}$.

The *abstract environment* w.r.t. P_i is the proclet $\overline{P_i} = (N, \mathit{ports})$ with $N = (\emptyset, \overline{T_i}, \emptyset)$, and $\mathit{ports} = \{q \mid (p, q) \in C_i \cup C_i^{-1}, q \notin \mathit{ports}_i\}$. The *abstracted system* $P_i \oplus \overline{P_i}$ is $(\{P_i, \overline{P_i}\}, C_i)$.

4.3 Behavioral Viewpoint: Extending Cases to Interaction Cases

We want to leverage existing conformance checking techniques to check conformance of a proclet system. As said before, our approach decomposes the problem of checking a proclet system into checking the conformance of each single proclet P_i and its abstract environment $\overline{P}_i$. For this, each case of P_i that is stored in the instance-aware log $\mathcal{L}$ needs to be extended to an *interaction case* by inserting all events of $\mathcal{L}$ that correspond to transitions of $\overline{P}_i$, and that send or receive messages from the instance *id* of this case.

Definition 12 (Interaction case, interaction log)**.** Let $\mathcal{L} = (\{L_1, \dots, L_n\}, <)$ be an instance-aware log. Let P_i be a proclet of a proclet system $\mathcal{P} = P_1 \oplus \dots \oplus P_n$, $i \in \{1, \dots, n\}$. Without loss of generality we select proclet P_i and its corresponding log L_i. Let $\sigma \in L_i$ be a case of an instance id of P_i. Let $\mathcal{E}$ be the set of all events in all cases in $\mathcal{L}$.

For each event $e = (a, id', SID, RID) \in \mathcal{E}$, we define $e|_{id}$ as follows:

- if $id' = id$, then $e|_{id} := e$ if $id' = id$;
- if $id \in SID$ or $id \in RID$, then $e|_{id} := (a, id', SID', RID')$ with $SID'(id) = SID(id)$ and $RID'(id) = RID(id)$ and $SID(id'') = RID(id'') = 0$, for all other $id'' \neq id$ (i.e., restrict communication of event e to communication with id);
- otherwise $e|_{id} := \perp$ (i.e., event e is *undefined* for instance id).

The set $\mathcal{E}|_{id} = \{e|_{id} \mid e \in \mathcal{E}, e|_{id} \neq \perp\}$ is the set of all events related to instance id. The *interaction case* of σ is the sequence $\overline{\sigma}$ containing all events $\mathcal{E}|_{id}$ ordered by $<$ of $\mathcal{L}$, i.e., $e < f$ implies $e|_{id} < f|_{id}$.

The *interaction log* of P_i w.r.t. $\mathcal{L}$ is the set $\mathcal{L}|_{P_i} := \{\overline{\sigma} \mid \sigma \in L_i\}$ containing the interaction case of each case of P_i in $\mathcal{L}$.

For example, the interaction cases $\overline{\sigma_{o1}}$ of σ_{o1} and $\overline{\sigma_{o2}}$ of σ_{o2} of the order proclet presented in Section 2.2 are

$$\begin{aligned}
\overline{\sigma_{o1}}: \quad & \langle(\mathsf{accept}, q_1, [], [o_1]), (\mathsf{accept}, q_2, [], [o_1]), (\mathsf{add\ CD}, o_1, [q_1], []),\\
& (\mathsf{add\ CD}, o_1, [q_2], []), (\mathsf{order}, o_1, [], []), (\mathsf{ship}, o_1, [], [q_1, q_2]), (\mathsf{processed}, q_1, [o_1], []),\\
& (\mathsf{processed}, q_2, [o_1], []), (\mathsf{close}, o_1, [], [])\rangle\\
\overline{\sigma_{o2}}: \quad & \langle(\mathsf{accept}, q_2, [], [o_2, o_2]), (\mathsf{add\ CD}, o_2, [q_2], []), (\mathsf{add\ CD}, o_2, [q_2], []), (\mathsf{order}, o_2, [], []),\\
& (\mathsf{notify}, o_2, [], [q_2]), (\mathsf{notified}, q_2, [o_2], []), (\mathsf{ship}, o_2, [], [q_2]), (\mathsf{processed}, q_2, [o_2], []),\\
& (\mathsf{close}, o_1, [], [])\rangle.
\end{aligned}$$

Note that in comparison to the original cases of Section 2.2, environment events of order such as accept changed the sets of expected senders and intended recipients, e.g., $(\mathsf{accept}, q_2, [], [o_1, o_2, o_2])$ became $(\mathsf{accept}, q_2, [], [o_1])$. The abstracted proclet system $\mathsf{quote} \oplus \overline{\mathsf{quote}}$ can replay these interaction cases.

4.4 The Decomposition is Correct

Decomposing a proclet system $\mathcal{P} = P_1 \oplus \dots \oplus P_n$ into abstracted proclet systems $P_i \oplus \overline{P_i}$ and replaying the interaction log of P_i on $P_i \oplus \overline{P_i}$, for each $i = 1, \dots, n$ equivalently preserves the fitness of $\mathcal{P}$ w.r.t. the given instance-aware event log $\mathcal{L}$.

Recall from Section 2.3.3 that $\mathcal{L}$ is replayed on $\mathcal{P}$ by ordering all events of $\mathcal{L}$ in a single case σ.

Definition 13 (Global case)**.** Let $\mathcal{L} = (\{L_1, \dots, L_n\}, <)$ be an instance-aware log. Let $\mathcal{E}$ be the set of all events in all cases in $\mathcal{L}$. The *global case* of $\mathcal{L}$ is the instance-aware case $\sigma_{\mathcal{L}}$ that contains all events $\mathcal{E}$ ordered by $<$.

With this notion, we can relate the replay behavior of the complete proclet system $\mathcal{P} = P_1 \oplus \ldots \oplus P_n$ to the replay behavior of the abstracted proclet systems $P_i \oplus \overline{P_i}$.

Theorem 1. *Let $\mathcal{P} = P_1 \oplus \ldots \oplus P_n$ be a proclet system and let $\mathcal{L}$ be an instance aware log. The global case $\sigma_{\mathcal{L}}$ can be replayed on $\mathcal{P}$ iff for all $i = 1, \ldots, n$, and each case $\sigma \in L_i$, the interaction case $\overline{\sigma}$ can be replayed on $P_i \oplus \overline{P_i}$.*

The first step to prove the correctness of the decomposition is a small lemma. From Definition 12 follows that we obtain each interaction case $\overline{\sigma^{id}}$ of an instance id of a proclet P_i also by projecting σ onto events that occur in id or exchange messages with id.

Lemma 1. *Let $\mathcal{L} = (\{L_1, \ldots, L_n\}, <)$ be an instance-aware log, let $\sigma_{\mathcal{L}}$ be the global case of $\mathcal{L}$. Let P_i be a proclet of a proclet system $\mathcal{P} = P_1 \oplus \ldots \oplus P_n$, $i \in \{1, \ldots, n\}$, and let $\sigma \in L_i$ be a case of an instance id of P_i.*

Let $\mathcal{E}$ be the set of all events in all cases in $\mathcal{L}$, and let $\mathcal{E}|_{id} = \{e|_{id} \mid e \in \mathcal{E}, e|_{id} \neq \bot\}$ be the set of all events related to instance id, as defined in Def. 12.

The sequence $\sigma_{\mathcal{L}}|_{id}$ obtained from $\sigma_{\mathcal{L}}$ by replacing each event e in $\sigma_{\mathcal{L}}$ with $e|_{id}$ and then removing all undefined events $\bot$ is the interaction case $\sigma_{\mathcal{L}}|_{id} = \overline{\sigma}$ of σ.

Proof. The proposition follows straight from Def. 12 and Def. 13. □

With leverage this lemma to configurations of the global case $\sigma_{\mathcal{L}}$ and of each interaction case $\sigma_{\mathcal{L}}|_{id}$.

Lemma 2. *Let $\sigma_{\mathcal{L}} = \langle e_1, e_2, e_3, \ldots, e_{n-1}, e_n \rangle$ be a global case that can be replayed on $\mathcal{P}$. Let $K_0 \ldots K_{n-2} \xrightarrow{e_{n-1}} K_{n-1} \xrightarrow{e_n} K_n$ be the run of $\mathcal{P}$ up to e_n.*

Let $\overline{\sigma} = \sigma_{\mathcal{L}}|_{id} = \langle f_1, f_2, \ldots, f_m \rangle$ be the interaction case of an instance id of a proclet P holds. Let $K'_0 \ldots K'_{m-2} \xrightarrow{f_{m-1}} K'_{m-1} \xrightarrow{f_m} K_m$ be the run of $P \oplus \overline{P}$ up to f_m.

The configurations $K'_m = (\mathcal{I}', m'_S, m'_C)$ reached by $\overline{\sigma}$ and $K_n = (\mathcal{I}, m_S, m_C)$ coincide on id:

1. *$id \in \mathcal{I}$ and $id \in \mathcal{I}'$,*
2. *for all places $s \in S_P$ in proclet P, $m_S(s)(id) = m'_S(s)(id)$, and*
3. *for all channels $c = (p, q) \in C$ s.t. p or q is a port of P, $[(id', id'') \in m_C(c) \mid id' = id \vee id'' = id] = [(id', id'') \in m'_C(c) \mid id' = id \vee id'' = id]$.*

Proof. We prove the lemma by induction on the length n of the global case $\sigma_{\mathcal{L}}$.

Consider the prefix $\overline{\sigma}' = \langle f_1, \ldots, f_r \rangle$ of $\overline{\sigma}$ that does not contain the last event $e_n|_{id}$, i.e., the last event e_n of $\sigma_{\mathcal{L}}$ is not part of $\overline{\sigma}'$. By inductive assumption, $\langle e_1, e_2, e_3, \ldots, e_{n-1} \rangle$ can be replayed on $\mathcal{P}$ reaching configuration K_{n-1} and $\overline{\sigma}'$ can be replayed on $P \oplus \overline{P}$ reaching configuration K' s.t. K_{n-1} and K' coincide on id.

If $\overline{\sigma}' = \overline{\sigma}$, i.e., $e_n|_{id} = \bot$ is not part of $\overline{\sigma}$, then $\overline{\sigma}$ can be replayed on $P \oplus \overline{P}$ (by inductive assumption), $K' = K'_m$, and hence the resulting configurations K'_m and K_n

coincide because e_n does not consume an id-colored token or a message sent to/received from id (by Def. 12 and Def. 10).

Otherwise, $\overline{\sigma} = \langle f_1, f_2, \ldots, f_r, e_n|_{id}\rangle$ where $\overline{\sigma}' = \langle f_1, f_2, \ldots, f_r\rangle$ reaches configuration $K' = K'_{m-1}$. Let $e_n = (t, id', RID, SID)$. We distinguish three cases:

1. $id = id'$, i.e., e_n occurs in id. Thus, $e_n|_{id} = e_n$. As K'_{m-1} and K_{n-1} coincide on id, transition t is enabled in id at K_{n-1} w.r.t. RID and SID iff t is enabled in id at K_{n-1} w.r.t. RID and SID (by Def. 9). From Def. 10 then follows that $K_{n-1} \xrightarrow{e_n} K_n$ and $K'_{m-1} \xrightarrow{e_n|_{id}} K'_m$ are steps of $\mathcal{P}$ and $P \oplus \overline{P}$, and K_n and K'_m coincide on id.

2. $id \in RID$, i.e., e_n sends messages to id and $e_n|_{id} = (t, id', SID', RID')$ where $RID(id) = RID'(id)$ (Def. 12). Moreover, t is a transition that produces messages into a channel reaching proclet P. Hence t is a transition of $\overline{P}$ and unrestricted in $P \oplus \overline{P}$. Thus, t is enabled at K'_{m-1} of $P \oplus \overline{P}$, and the steps $K_{n-1} \xrightarrow{e_n} K_n$ and $K'_{m-1} \xrightarrow{e_n|_{id}} K'_m$ produce the same number of messages to id in channel c (by Def. 10). So K_n and K'_m coincide on id.

3. $id \in SID$, i.e., e_n expects messages from id and $e_n|_{id} = (t, id', SID', RID')$ where $SID(id) = SID'(id)$ (Def. 12). Moreover, t is a transition that consumes messages from a channel c leaving proclet P. As K'_{m-1} and K_{n-1} coincide on id, the channel c contains the same number of messages sent from id in both configurations. Thus, t is enabled at K'_{m-1} of $P \oplus \overline{P}$, and the steps $K_{n-1} \xrightarrow{e_n} K_n$ and $K'_{m-1} \xrightarrow{e_n|_{id}} K'_m$ consume the same number of messages sent by id from channel c (by Def. 10). So K_n and K'_m coincide on id.

This proves the induction hypothesis that configurations of the global case $\sigma_{\mathcal{L}}$ and the configurations of each interaction case $\sigma_{\mathcal{L}}|_{id}$ of a proclet instance id coincide on id. $\square$

As a consequence, we obtain the following corollary.

Corollary 1. *Let $\sigma_{\mathcal{L}}$ be a global case that can be replayed on $\mathcal{P}$. Then for each proclet P of $\mathcal{P}$, and each instance id of P, the interaction case $\sigma_{\mathcal{L}}|_{id}$ of id can be replayed on $P \oplus \overline{P}$.*

We can now prove Theorem 1.

Proof of Theorem 1. ($\Rightarrow$) Let $\sigma_{\mathcal{L}}$ be a global case that can replayed on $\mathcal{P}$. Then Corollary 1 already states that each interaction case $\sigma_{\mathcal{L}}|_{id}$ of a proclet instance id of a proclet P of $\mathcal{P}$ can be replayed on $P \oplus \overline{P}$.

($\Leftarrow$) Let $\sigma_{\mathcal{L}}$ be a global case that cannot be replayed on $\mathcal{P}$. Let $\sigma'_{\mathcal{L}}$ be the largest prefix of $\sigma_{\mathcal{L}}$ that still can be replayed, reaching configuration K. Let $e = (t, id, SID, RID)$ be the first event after $\sigma'_{\mathcal{L}}$ that cannot be replayed in $\mathcal{P}$, i.e., t is not enabled in id at K w.r.t. SID and RID.

Proposition: t is not enabled at the end of the interaction case $\sigma'_{\mathcal{L}}|_{id}$ of instance id.

Let K' be the configuration reached at the end of $\sigma'_{\mathcal{L}}|_{id}$; K and K' coincide on id by Lemma 2. According to Def. 9, there are three possible reasons for t not being enabled:

1. t is not enabled at K because one of its pre-places does not contain enough tokens in instance id. Lemma 2 implies that, t of P is not enabled at K', and $\sigma'_{\mathcal{L}}|_{id}$ cannot be replayed on $P \oplus \overline{P}$.

2. t is not enabled at K because there are insufficiently many messages in the channel. Like in the preceding case, Lemma 2, implies that K' contains the same number of messages. As all ports of proclet P remain the same, t is also not enabled at K' of $P \oplus \overline{P}$.

3. t is not enabled at K because RID is not valid, i.e., the number of intended recipients violates the port constraint if the port p of proclet P to which t is attached. This port p is the same in $P \oplus \overline{P}$, so RID is not valid in t either and t is not enabled at K'.

Thus, the case $\sigma'_{\mathcal{L}}|_{id}$ cannot be replayed on $P \oplus \overline{P}$, where id is an instance of proclet P. □

5 Conformance Checking Techniques for Artifacts

The previous transformations of abstracting a proclet's environment and extracting interaction cases allow us to isolate a single proclet instance for conformance checking w.r.t. the proclet and its associated channels. In other words, we reduced artifact conformance to the problem of checking whether the proclet system $P \oplus \overline{P}$ can replay the interaction log $\mathcal{L}|_P$, where each case in $\mathcal{L}|_P$ only refers to exactly on proclet instance. Thus, the problem can be fed into existing conformance checkers.

Our conformance checker leverages the technique described in [3]. As this technique only takes Petri nets and classical cases (without instance-aware events) as input, the conformance checking problem of $P \oplus \overline{P}$ w.r.t. $\mathcal{L}|_P$ needs to be further reduced to a conformance checking problem on Petri nets w.r.t. a normal, non-instance aware log.

1. Our reduction translates the proclet ports into *Petri net patterns* that have the same semantics. Replacing each port of P in $P \oplus \overline{P}$ with its respective pattern yields a Petri net N_P that equivalently replays the interaction cases $\mathcal{L}|_P$; the details of this translation are given in Section 5.1.
2. Classical conformance checking techniques are not aware of interaction cases (Def. 12). In particular, it is unaware that event $e = (a, id', [id, id], [])$ sends two messages from instance id' to instance id. For this reason, each interaction case needs to be transformed in a classical case (without instances) in a way that guarantees that messages sent to id and sent from id are preserved. We show this transformation in Section 5.2.

5.1 Mapping from Proclets to Reset Nets

The transformations of abstracting a proclet's environment (Section 4.2) and extracting interaction cases (Section 4.3) allow to isolate a single proclet instance for conformance checking. Yet, the formal model is still a proclet which cannot be fed into existing conformance checking techniques as these are not aware of the proclet's ports in their semantics. This section provides a translation of $P_i \oplus \overline{P}_i$, the proclet P_i with its abstract environment $\overline{P}_i$, to a reset net N_i so that N_i conforms to a trace σ_{id} iff $P \oplus \overline{P}$ conforms to σ_{id}.

The translation primarily provides a semantics for proclet ports in terms of *inhibitor-reset nets* (or IR-nets), an extension of Petri nets with *inhibitor arcs* and *reset arcs.* A reset arc allows to consume *all* tokens in a place instead of just one; an inhibitor arc prevents the occurrence of a transition if the connected place has a token.

We first define inhibitor-reset nets and their semantics, then present a set of *translation patterns* that give for each combination of cardinalities and multiplicities of a port p a corresponding reset net pattern, and then show how to apply these patterns in a small example.

Definition 14 (Reset net)**.** An *inhibitor reset net* (or IR-net) $N = (S, T, F, I, R, m_0)$ is a Petri net (S, T, F) extended by a set $I \subseteq (P \times T)$ of *inhibitor arcs*, a set $R \subseteq (P \times T)$ *reset arcs*, and an *initial marking* $m_0 : S \to \mathbb{N}$ that assigns each place s a natural number $m_0(s)$ of tokens.

We depict an inhibitor arc with a filled circle instead of an arrow head, and a reset arc with a double arrow head as shown in Fig. 5.1. Note that a place s and a transition t may be connected by a normal arc $(s, t) \in F$ as well as a reset arc $(s, t) \in R$. Connecting s and t with a normal arcs and an inhibitor arc is also allowed, but would be contradictory as we defined next.

The semantics of an IR-net is essentially the semantics of a Petri net where a transition t is only enabled if no place s with an inhibitor arc (s, t) contains a token; a reset arc (s, t) additionally removes *all* tokens in s when t occurs.

Definition 15 (Semantics of a reset net)**.** Let $N = (S, T, F, I, R, m_0)$ be a reset net and let $m : S \to \mathbb{N}$ be a marking of N. Transition $t \in T$ is *enabled* iff $m(s) > 0$, for each $(s, t) \in F$, and $m(s) = 0$, for each $(s, t) \in I$. If t is enabled, t can *occur* defining the *step* $m \xrightarrow{t} m_t$ of N that reaches the successor marking m_t of m as follows:

1. compute an intermediate marking m' with $m'(s) = m(s) - 1$ if $(s, t) \in F \setminus R$, $m'(s) = 0$ if $(s, t) \in R$, and $m'(s) = m(s)$ otherwise; and
2. set $m_t(s) = m'(s) + 1$ if $(t, s) \in F$, and $m_t(s) = m'(s)$ otherwise.

A *sequential run* of N is a sequence $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \xrightarrow{t_3} \ldots$ of steps of N starting in the initial marking m_0.

Inhibitor reset net patterns for ports. Figures 5.1 and 5.2 define an IR-net pattern for each possible port of a proclet. Each respective pattern replaces the port p attached to a transition with label x when translating $P_i \oplus \overline{P_i}$ to a reset net N_i.

The patterns are constructed in a modular way. Each patterns replaces port p by a place p. In addition, each row and each column of Figures 5.1 and 5.2 defines a distinctive feature to express the cardinality constraint c and the multiplicity constraint m of p, respectively. The features are combined to define the semantics of p. The respective features are the following.

Input ports and output ports define multiplicity constraints in the same way.

- $m = 1$ requires action x to occur exactly once. Thus, all transitions with label x get a new shared pre-place s_x that is initially marked with one token. This token must be consumed which is expressed by the inhibitor arc from s_x to the final transition of the proclet, i.e., the token must be consumed to let final occur.

- $m = ?$ requires action x to occur at most once. Thus, all transitions with label x get a new shared pre-place s_x that is initially marked with one token. If the token is not consumed (no x occurred), then the final action will remove this token via the reset arc.
- $m = +$ requires action x to occur at least once. Thus, all transitions with label x get a new shared post-place s_x that needs to be marked (x occurs) to enable the final action. The final action will remove all produced tokens to clean up the net.
- $m = *$ makes no constraints regarding occurrences of x.

The cardinality constraints govern how many messages to send or to receive and hence are formalized slightly differently in the patterns. We begin with the patterns for input ports shown in Fig. 5.1.

- $c = 1$ requires action x to consume one message, which translates to a simple arc from place p to each transition attached to the port.
- $c = ?$ allows action x to occur with or without consuming a message from p. Thus, the pattern for $c = 1$ is extended by a new transition with label x that does not consume from p.
- $c = +$ requires action x to consume all messages from p (and at least one such message). Thus, the pattern for $c = 1$ is extended by a reset-arc from p to each transition attached to the port.
- $c = *$ allows action x to occur by consuming an arbitrary number of messages from p, which is expressed by a reset arc from place p to each transition attached to the port.

The patterns for output ports shown in Fig. 5.2 are symmetric with the directions of the arcs attached to place p being reversed. However, this implies inverting the notion of the reset arcs which removes an arbitrary number of tokens into an arc that produces an arbitrary number of tokens. The graphical representation of Fig. 5.2 is only a short hand notation for a pattern shown in Fig. 5.3. According to these patterns, a single "inverse reset arc" from transition t to place p is replaced by a new intermediate post-place p_t of t and two internal transitions. The first internal transition allows to produce an arbitrary number of tokens on p as it has p_t in its pre- and post-set. The second internal transition consumes from p_t and produces on the original post-place s of t. The pattern for an "inverse reset arc" from t to p together with a normal arc from t to p defines an additional arc from the second internal transition to p which ensures that at least on message will be produced. Fig. 5.3 to the right shows for the example of $c = +, m = +$ how to combine this pattern with the other patterns defined in Fig. 5.2.

The patterns for input ports have an additional inhibitor arc from the place p representing the port to the final transition. This arc expresses that no messages for a particular proclet instance remain pending in the channel when the instance finishes its life-cycle. For output ports, pending messages are checked in the proclet at the other end of the channel, as this is responsible for consuming incoming messages.

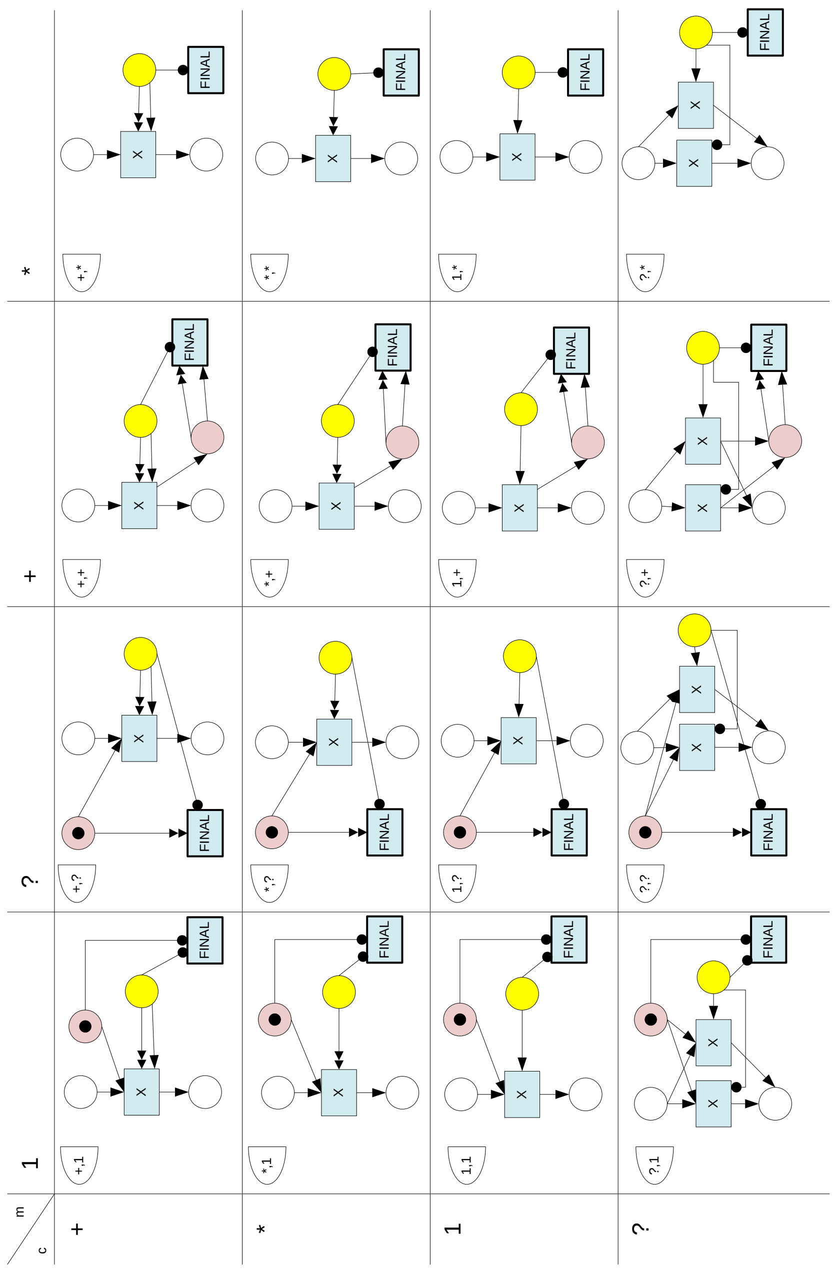


Figure 5.1: Patterns for translating a transition with label x attached to an input port p to reset nets.

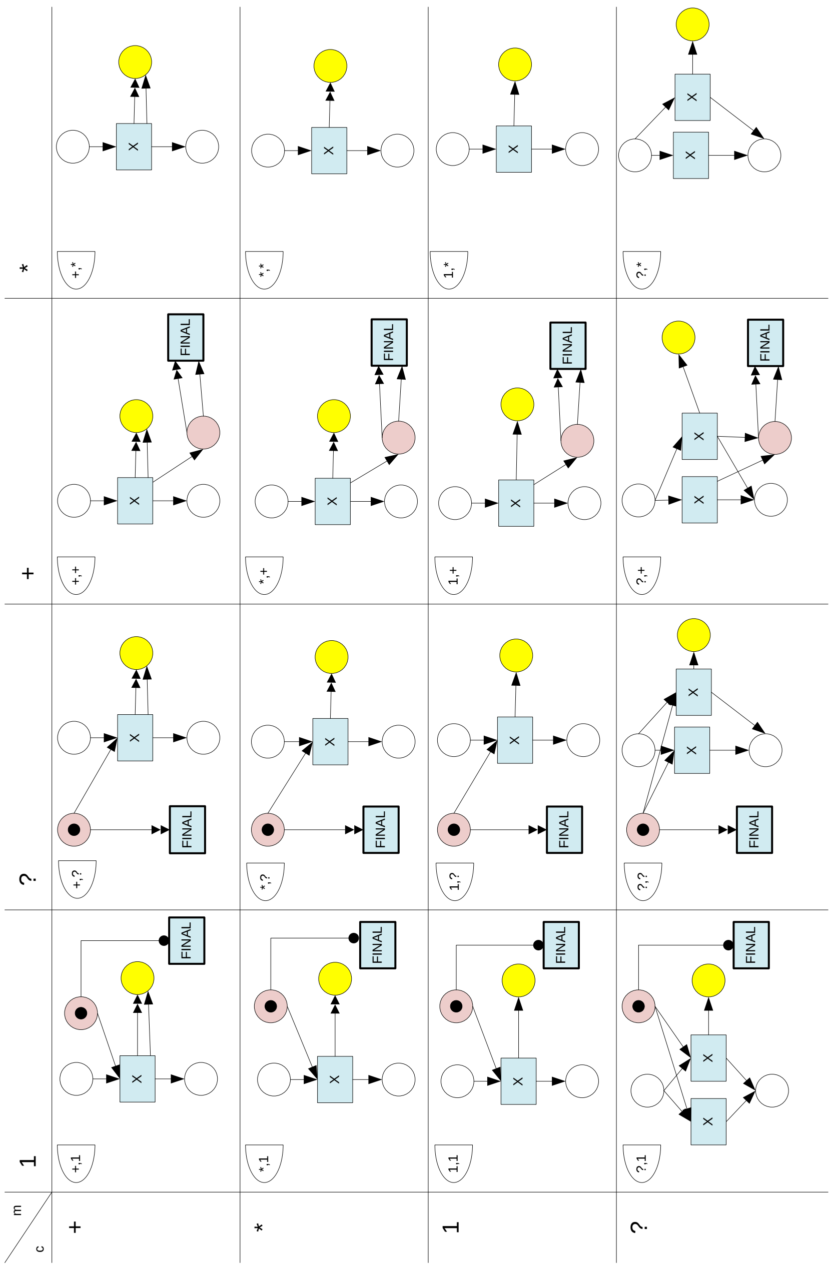


Figure 5.2: Patterns for translating a transition with label x attached to an output port p to reset nets. The patterns with cardinality $+$ and $*$ use a short-cut notation (arc with two arrow heads) to express the sending of multiple messages; Fig. 5.3 defines the short-cut notation.

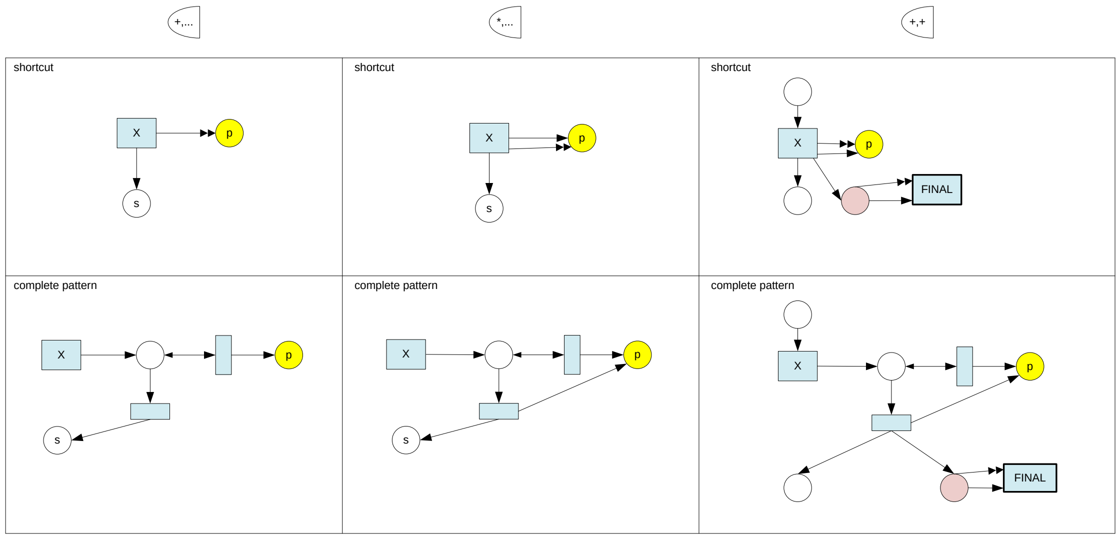


Figure 5.3: Shortcuts used in the translation of output ports (Fig. 5.2) to modeling sending multiple message.

Applying the reset net patterns, an example. The following example illustrates the transformation of a proclet P_i with its abstract environment $\overline{P}_i$ to a reset net N_i.

1. First, translate P_i to a net N_i using the patterns shown in Figures 5.1, 5.2, and 5.3.

2. Then extend N_i by the transitions of $\overline{P}_i$ and new arcs as follows. If transition t of $\overline{P}_i$ is attached to a port q with a channel (q, p) of $P_i \oplus \overline{P}_i$, then add an arc from t to the place p of N_i that represents the input port p of P_i. Conversely, if t is attached to a port with a channel (p, q) of $P_i \oplus \overline{P}_i$, then add an arc from the place p of N_i that represents the output port p of P_i to transition t.

The resulting net N_i conforms to the same traces as $P_i \oplus \overline{P}_i$. Figure 5.4 shows the result of translating $\mathsf{quote} \oplus \overline{\mathsf{quote}}$ of Fig. 2.1 to an IR-net N_{quote}. The dashed boxes in Fig. 5.4 are only used to illustrate the origin of the respective places and arcs, they have no semantic meaning.

5.2 Mapping Interaction Cases to Cases of Inhibitor-Reset Nets

Classical conformance checkers, such as the one we want to use to solve interaction conformance of artifacts [3], take as input a Petri net (or an inhibitor reset net in our case), and a *classical case* being a plain sequence of event types $\sigma \in \Sigma^*$ (see Def. 1). Section 5.1 translated a proclet with its environment $P \oplus \overline{P}$ to an IR-net N_P. Correspondingly, we have to translate each interaction case of P (being a sequence of instance-aware events) to a classical case (being a sequence of event types).

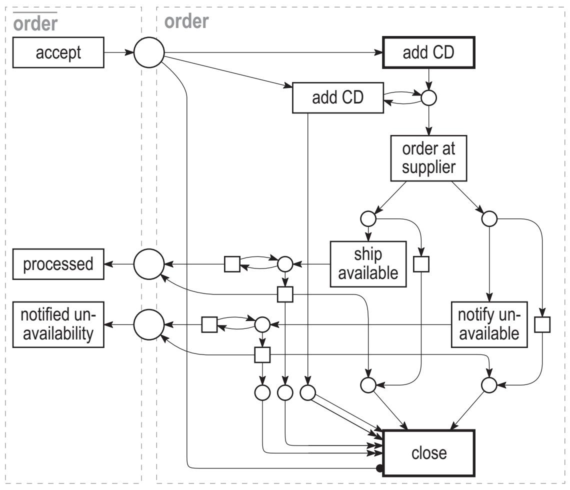


Figure 5.4: The result of translating $\mathsf{quote} \oplus \overline{\mathsf{quote}}$ of Fig. 2.1 to an IR-net N_{quote}.

The particular difference is the number of messages being generated by the environment $\overline{P}$ for P. A transition t of the proclet $\overline{P}$ that is attached to a port p with cardinality $+$ or $*$ can produce several messages for P, i.e., t can replay an instance-aware event $e = (t, id', [id, id], [])$ which produces two messages for id. After the translation to N_P, the same transition can only replay the classical event e' being an occurrence of t, producing only one message for id.

To preserve the number of messages generated by t in $\overline{P}$, we transform each interaction case $\overline{\sigma}$ of P into a classical case σ_P by replacing each event $e = (t, id', SID, [])$ with $SID(id)$-many events describing occurrences of t. In other words, replace e by a sequence $t \dots t$ of length $SID(id)$. For example, the interaction case $\overline{\sigma_{o2}}$ (Sect. 4.3) is transformed to $\langle \mathsf{accept}, \mathsf{accept}, \mathsf{add\ CD}, \mathsf{add\ CD}, \mathsf{order}, \mathsf{notify}, \mathsf{notified}, \mathsf{ship}, \mathsf{processed}, \mathsf{close} \rangle$.

5.3 Replay Techniques for Inhibitor-Reset Nets

After converting the proclets into Petri Nets $N_{P_1}, \dots, N_{P_n}$ and translating their interaction cases as mentioned above, our conformance checker applies the technique of [3] to check how good the net N_{P_i} replays $\mathcal{L}|_{P_i}$, for each $i = 1, \dots, n$ separately. Technically, the checker finds for each interaction case $\sigma \in \mathcal{L}|_{P_i}$ an execution σ' of N_{P_i} that is as *similar* as possible to t. If N_{P_i} cannot execute σ, then σ is changed to an execution σ' of N_{P_i} by inserting or removing actions of N_{P_i}. The more σ' deviates from σ, the less N_{P_i} fits σ. The fitness of N_{P_i} on σ is defined by a cost-function that assigns a penalty on σ' for each event that has to be added or removed from σ to obtain σ'. The most similar σ' is found by efficiently exploring the search space of finite sequences of actions of N_{P_i} guided by the cost function [3]. The fitness of N_{P_i} w.r.t. L_{P_i} is the average fitness of N_{P_i} w.r.t. all cases in L_{P_i}.

The fitness of the entire proclet system $P_1 \oplus \dots \oplus P_n$ w.r.t. $\mathcal{L}$ is the average of the fitness of each P_i to its interaction cases $\mathcal{L}|_{P_i}$.

To illustrate the misconformances that can be discovered with this technique, assume that in the process execution of Fig. 2.2, case q_1 did not contain an accept event. This would lead to the following interaction case of o_1:

$$\begin{aligned}&\langle(\mathsf{accept}, q_2, [], [o_1, o_2, o_2]), (\mathsf{add\ CD}, o_1, [q_1], []),\\&(\mathsf{add\ CD}, o_1, [q_2], []), (\mathsf{order}, o_1, [], []), (\mathsf{ship}, o_1, [], [q_1, q_2]), (\mathsf{processed}, q_1, [o_1], []),\\&(\mathsf{processed}, q_2, [o_1, o_2], []), (\mathsf{close}, o_1, [], [])\rangle.\end{aligned}$$

Our conformance checker would then detect that the cardinality constraint + of the input port of add CD would be violated: only one message is produced in the channel, but two occurrences of add CD are noted, each requiring one message.

6 Operationalization in ProM

The *interaction conformance checker* is implemented as a software plug-in of ProM, a generic open-source framework and architecture for implementing process mining tools in a standard environment [18]. The interaction conformance checker plug-in takes as input the proclet system model and the projection of the log with respect to a proclet of interests (e.g., the order proclet) and, by employing the techniques described in Section 4, returns an overview of the deviations between the cases in the log and the proclet system model.

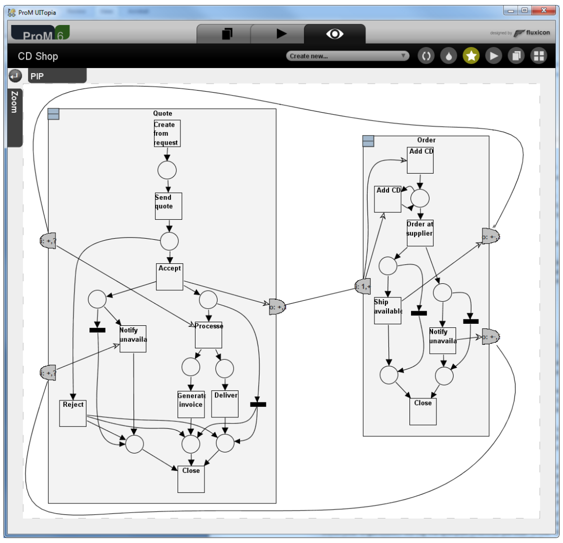
As input for our initial experiments, we generated synthetic event logs using CPN Tools (`http://cpntools.org`) and we manually created the projections to the respective artifacts. The proclet system was hard-coded in ProM and we implemented generic conversions from proclet systems (called artifact models in ProM) to Petri net-based versions of proclets with their environments. More details are given in the report [6].

For the interaction conformance, an existing conformance checker was used [2], which is capable of replaying logs on Petri nets that include reset- and inhibitor arcs which are needed for expressing the replay semantics of proclets.

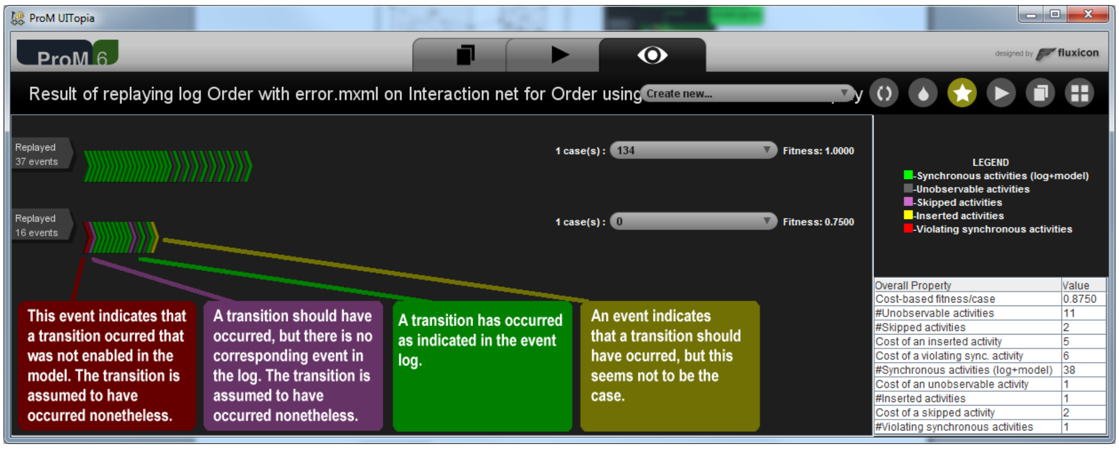
Figure 6.1(a) shows how the model of the *CD shop* example is visualized in ProM, where every light-gray rectangle is a proclet. Visible transitions are represented as empty squares, whereas small filled rectangles are the invisible transitions.

The result of the conformance checking is shown in Figure 6.1(b). For clarity, we show a log with only two cases, one conforming case one deviating case. Every row identifies a different case in which the execution replay is represented as a sequence of wedges. Every wedge corresponds to *(a)* a "move" in both the model and the log, *(b)* just a "move" in the model (skipped transition), or *(c)* just a "move" in event log (inserted event).

For a case without any problems, i.e., just moves of type *(a)*, fitness is 1. The first case in Figure 6.1(b) has fitness 1. Note that the conformance checker identified some non-labeled transitions to have fired (indicated by the black triangles). These are the transitions necessary to model cardinality of the ports. The second case shows a lower conformance. The conformance checker identifies where the case and the model disagree and, using a color coding, it shown exactly what type of deviation occurs. This information allows us to understand to what extent the model used to express the process fits the actual performance experienced in reality and where deviations are shown.



(a) The model of the CD shop example



Result of replaying log Order with error.mxml on Interaction net for Order using

Fitness: 1.0000

Fitness: 0.7500

This event indicates that a transition occurred that was not enabled in the model. The transition is assumed to have occurred nonetheless.

A transition should have occurred, but there is no corresponding event in the log. The transition is assumed to have occurred nonetheless.

A transition has occurred as indicated in the event log.

An event indicates that a transition should have ocurred, but this seems not to be the case.

LEGEND

🟩-Synchronous activities (log+model)
⬛-Unobservable activities
🟪-Skipped activities
🟨-Inserted activities
🟥-Violating synchronous activities

Overall Property	Value
Cost-based fitness/case	0.8750
#Unobservable activities	11
#Skipped activities	2
Cost of an inserted activity	5
Cost of a violating sync. activity	6
#Synchronous activities (log+model)	38
Cost of an unobservable activity	1
#Inserted activities	1
Cost of a skipped activity	2
#Violating synchronous activities	1

(b) The conformance results for the *order* proclet

Figure 6.1: Screenshots in ProM

7 Related Work

There are many metrics that quantify the extent a given process execution conforms to a given model. These range from naïve metrics that simply count the fraction of cases that can be fully executed in a model [19, 9] to more advanced metrics looking at a more fine-grained level. A comprehensive list of existing conformance metrics can be found in [15].

The more advanced conformance metrics are able to reflect the fact that in some traces only parts are deviating [19, 11], but also pinpoint where deviations occur [14], while taking into account the fact that models may contain behavior that is unobservable by nature [3]. This allows these metrics to be applied in the context of many process modeling languages, including proclets.

Several BPM researchers have investigated compliance at the model level [8]. These approaches do not take the observed behavior into account.

8 Conclusion

In this paper, we considered the problem of determining whether different artifacts interact according to their specification. We take the emerging paradigm of *artifact-centric* processes as a starting point. Here, processes are composed of interacting artifacts, i.e., data objects with a life-cycle. The paradigm allows for the modeling of many-to-many relationships and complex interactions.

Conformance checking techniques have been focusing on checking the conformance of one instance in isolation. In this paper, we lift the question to artifact-centric processes. We use proclets—one of the first artifact-centric notations—to investigate the problem.

In this paper, we showed that the problem of interaction conformance can be decomposed into a number of sub-problems for which we can use classical conformance checking techniques. More specifically, we show that we can look at each artifact in isolation if we include it's *environment* which consists of single transitions of the surrounding proclets. The approach is supported by ProM and could also be applied to other artifact-centric notations.

Future Work. In future work, we plan to extend this research in several directions. Firstly, we aim at an automatic extraction of interaction cases from recorded executions of an information system. The approach presented in this report needs to be implemented. However, in some cases, the behavioral information is not stored separate logs, one log per artifact, but only in the database that supports the process. The goal is to develop techniques for manual and automatic extraction of cases from structured databases. Another extension is to adapt further conformance metrics (Sect. 7) to the artifact setting.

A complimentary to technique to obtain accurate descriptions of processes executed in reality is *process mining* [1, 17]; it automatically discovers a process model from recorded system executions. The open problem that we want to address is to leverage process mining techniques from the classical setting of isolated process instances to the artifact setting of interacting and overlapping instances.

Bibliography

[1] W.M.P. van der Aalst, H.A. Reijers, A.J.M.M. Weijters, B.F. van Dongen, A.K. Alves de Medeiros, M. Song, and H.M.W. Verbeek. Business Process Mining: An Industrial Application. *Information Systems*, 32(5):713–732, 2007.

[2] A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. Conformance Checking using Cost-Based Fitness Analysis. Technical report, BPMcenter.org, 2011. BPM Center Report (submitted).

[3] Arya Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. Towards Robust Conformance Checking. In *BPM'10 Workshops*, 2010. LNBIP to appear.

[4] Alistair P. Barros, Gero Decker, Marlon Dumas, and Franz Weber. Correlation patterns in service-oriented architectures. In *FASE*, volume 4422 of *LNCS*, pages 245–259. Springer, 2007.

[5] David Cohn and Richard Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.*, 32(3):3–9, 2009.

[6] Dirk Fahland, Massimiliano de Leoni, Boudewijn van Dongen, and Wil van der Aalst. Artifact process conformance checking. Technical report, Eindhoven - Technical University of Technology, March 2011.

[7] Christian Fritz, Richard Hull, and Jianwen Su. Automatic construction of simple artifact-based business processes. In *ICDT'09*, volume 361 of *ACM ICPS*, pages 225–238, 2009.

[8] G. Governatori, Z. Milosevic, and S. W. Sadiq. Compliance Checking Between Business Processes and Business Contracts. In *EDOC 2006*, pages 221–232. IEEE Computer Society, 2006.

[9] G. Greco, A. Guzzo, L. Pontieri, and D. Sacca. Discovering Expressive Process Models by Clustering Log Traces. *IEEE Trans. on Knowl. and Data Eng.*, 18:1010–1027, August 2006.

[10] Niels Lohmann and Karsten Wolf. Artifact-centric choreographies. In *ICSOC 2010*, volume 6470 of *LNCS*, pages 32–46. Springer, December 2010.

[11] A.K. Alves de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Genetic Process Mining: An Experimental Evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304, 2007.

[12] A. Nigam and N.S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[13] A. Rozinat, I.S.M. de Jong, C.W. Gunther, and W.M.P. van der Aalst. Conformance Analysis of ASML's Test Process. In *GRCIS'09*, volume 459 of *CEUR-WS.org*, pages 1–15, 2009.

[14] A. Rozinat and W.M.P. van der Aalst. Conformance Checking of Processes Based on Monitoring Real Behavior. *Information Systems*, 33(1):64–95, 2008.

[15] Anne Rozinat, Ana Karla Alves de Medeiros, Christian W. Günther, A. J. M. M. Weijters, and Wil M. P. van der Aalst. The Need for a Process Mining Evaluation Framework in Research and Practice. In *BPM'07 Workshops*, volume 4928 of *LNCS*, pages 84–89. Springer, 2007.

[16] Wil M. P. van der Aalst, Paulo Barthelmess, Clarence A. Ellis, and Jacques Wainer. Proclets: A Framework for Lightweight Interacting Workflow Processes. *Int. J. Cooperative Inf. Syst.*, 10(4):443–481, 2001.

[17] B.F. van Dongen, A.K. Alves de Medeiros, and L. Wen. Process Mining: Overview and Outlook of Petri Net Discovery Algorithms. *ToPNOC*, 2:225–242, 2009.

[18] H.M.W. Verbeek, Joos C.A.M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. ProM: The Process Mining Toolkit. In *BPM Demos 2010*, volume 615 of *CEUR-WS*, 2010.

[19] A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.