Conformance Checking of Interacting Processes With Overlapping Instances

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|w.m.p.v.d.aalst)@tue.nl

Abstract. 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. In particular, 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

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. [1]). To avoid building on quicksand it is vital to know in advance to what extent a 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 [2–7]. 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 match the reality recorded by S (or the system 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 [8–10].

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 process in reality — despite the absence of process instances.

The first problem is well-known [8–10] and several modeling languages have been proposed to solve it culminating in the stream of *artifact-centric process modeling* that emerged in recent years [8–12]. 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 [11, 12]. 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* [8] 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 an interface to other proclets. A *proclet system* connects the interfaces of its proclets via unidirectional channels, allowing the life-cycles of instances of the connected proclets to interact with each other by exchanging messages; one instance may send messages 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. For each event, it is recorded 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 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 objects, according to other cases in other logs. We then show how to abstract a given proclet system $\mathcal{P}$ to an abstract proclet system $\mathcal{P}|_P$ for each proclet P s.t. $\mathcal{P}$ can replay $L_1, \ldots, L_n$ iff the abstract proclet system $\mathcal{P}|_P$ can replay each interaction case of P, for each proclet P of $\mathcal{P}$. As an interaction case focuses on a single instance of a single proclet at a time (while taking its interactions into account), existing conformance checkers [6,7] can be used to check conformance.

This paper is structured as follows. Section 2 recalls how artifacts describe 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. The entire approach is implemented in the process mining toolkit ProM; Section 5 presents the tool's functionality and shows how it can discover deviations in artifact-centric processes. Section 6 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.

2.1 Artifacts and Proclets: 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 for 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. In the recent years, the *artifact-centric approach* emerged as a paradigm to describe processes like in our CD shop example

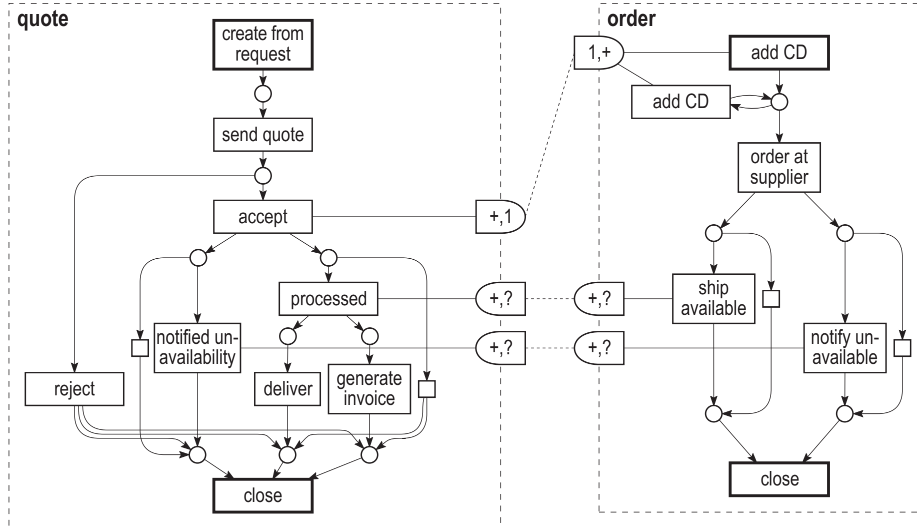

Fig. 1. A proclet system describing the back-end process of a CD online shop. A customer's quote is split into several orders according to the suppliers of the CDs; an order at a supplier handles several quotes from different customers.

where several cases of quotes object interact with several cases of orders. *Quotes* and *orders* are the *artifacts* of this process. Figure 1 models the above process in terms of a *proclet system* [8] consisting of two *proclets*: one describing the life-cycle of *quotes* and the other describing the life-cycle of *orders*. Note that Fig. 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 may interact with *several* orders, and each order may interact 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 execution gives rise to the following cases of quote and order which interact as illustrated in Fig. 2 (note that a CD is not an artifact as it does not follow a life-cycle in this process).

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

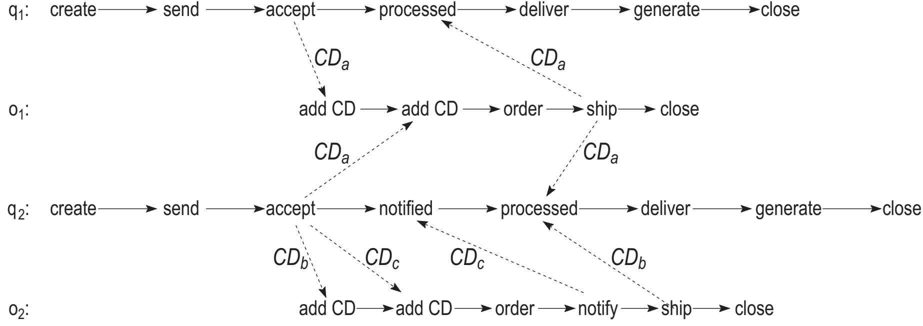

Fig. 2. An execution of the CD shop process involving two quote cases and two order cases that interact with each other in a many-to-many fashion.

2.2 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 case can be represented by the sequence of events that occurred. In an artifact-centric process like Fig. 1, one cannot abstract from interactions and many-to-many relations between different cases; 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 [13] 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 defines a *conversation*. For instance in Fig. 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 [13]. 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 illustrated in Fig. 2.

To abstract from a specific correlation mechanism we introduce the notion of an *instance-aware log*. In the following we assume asynchronous interaction between different instances. Let e be an event. Event correlation defined the instances from which e received a message and the instances to which e sent a message. As e could send/receive several messages to/from the same instance, correlation data are stored as *multisets* of instance ids. For sake of simplicity, in this paper we assume that correlation data on receiving and sending messages was defined by a single correlation property each. Hence, e is associated to *one multiset of instances from which* e *received messages* and to *one multiset of instances to which* e *sent messages.* A multiset $m \in \mathbb{N}^{\mathcal{I}}$ over a set $\mathcal{I}$ of instance ids is technically a mapping $m : \mathcal{I} \to \mathbb{N}$ 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 in which e *occurred,* $SID = [sid_1, \ldots, sid_k] \in \mathbb{N}^{\mathcal{I}}$ *is the multiset of instances from which* e *consumed messages, and* $RID = [rid_1, \ldots, rid_l] \in \mathbb{N}^{\mathcal{I}}$ *is the multiset of instances for which* e *produced messages.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. 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 $(\mathsf{processed}, q_2, [o_1, o_2], [])$. Instance-aware events capture the essence of event correlation and abstraction from the underlying correlation property, e.g., $\mathsf{CD_a}$.

All events of one instance of an artifact A define a case; all cases of A define the log of A. An execution of the entire process records the cases of the involved artifact 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 are the following:

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

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

2.3 Proclets

Different languages for describing artifacts have been proposed [8–12]. In the following, we use *proclets* [8] to study instantiation of artifacts and the many-to-many interactions between different artifact instances in a light-weight formal model. A proclet describes 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 specify how many instances interact with each other via a channel.

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*.*

[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 4 (Proclet). *A* proclet $P = (N, ports)$ *consists of a labeled Petri net* $N = (S, T, F, \ell)$ *and a set of ports, where*

- *some transition* $initial \in T$ *has no pre-place (i.e.,* $\{s \mid (s, initial) \in F\} = \emptyset$*) and some transition* $final \in T$ *has no post-place (i.e.,* $\{s \mid (final, s) \in F\} = \emptyset$*),*
- *each port* $p = (T^p, dir_p, card_p, mult_p)$ *is (1) associated to a set* $T^p \subseteq T$ *of transitions, has (2) a direction of communication* $dir_p \in \{in, out\}$ *(i.e., receive or send messages, resp.), (3) a* cardinality $card_p \in \{?, 1, *, +\}$*, and (4) a* multiplicity $mult_p \in \{?, 1, *, +\}$*, and*
- *each transition is attached to at most one input port and to at most one output port, i.e., for all* $t \in T$ *holds* $|\{p \in ports \mid t \in T^p, dir_p = in\}| \leq 1$, $|\{p \in ports \mid t \in T^p, dir_p = out\}| \leq 1$.

Instead of an initial marking a proclet has an initial transition which will produce the first tokens in the net. This will allow us to express the creation of multiple instances of the same proclet.

Figure 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). Although the example happens to be acyclic, proclets may contain cycles.

Definition 5 (Proclet system). *A* proclet system $\mathcal{P} = (\{P_1, \ldots, P_n\}, C)$ *consists of a finite set* $\{P_1, \ldots, P_n\}$[2] *of proclets together with a set* C *of channels s.t. each channel* $(p, q) \in C$ *connects an output port* p *to an input port* q*,* $p, q \in \bigcup_{i=1}^{n} \text{ports}_i$, $dir_p = out$, $dir_q = in$.

Without loss of generality, we assume the proclets' sets of transitions and places to be pairwise disjoint. Hence, the labeling of the proclets lifts to the proclet system: $\ell_{\mathcal{P}}(t) := \ell_i(t)$, for each transition $t \in T_i$ of each proclet $i = 1, \ldots, n$. We will also write $\mathcal{P} = P_1 \oplus \ldots \oplus P_n$ as a shorthand for $\mathcal{P} = (\{P_1, \ldots, P_n\}, C)$. Figure 1 shows the proclet system consisting of proclets quote and order. Extending the usual notation for Petri nets, each half-round shape represents a port; the bow indicates the direction of communication. A dashed line between 2 ports denotes a channel of the proclet system.

2.4 Semantics of Proclets: Overlapping Cases

During execution, there may be several instances of the same proclet running concurrently. Instances are created dynamically during process execution, that is, whenever there is a need for a new instance, one will be created. Initial and final transitions of a proclet (depicted in bold in Fig. 1) express instantiation and termination, i.e., whenever create of proclet quote occurs, a *new* instance of quote is created; the top-most transition add CD creates a new *order instance*.

[2] 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.

Proclet instances interact with each other by sending messages over the channels of the proclet system. A transition attached to an output port *sends* messages, a transition attached to an input port *receives* messages. A port p's annotations specify how many messages are sent or received (cardinality $card_p$) and how often the port can be used by a proclet instance to send or receive messages (multiplicity $mult_p$). For example, cardinality + of the port of accept denotes that a quote sends out one or more messages on quoted CDs to multiple orders per invocation. Its multiplicity 1 indicates that there is precisely one such invocation during the lifecycle of a quote. Conversely, the process repeatedly (multiplicity +) adds one CD of a quote to an order (cardinality 1).

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). So, altogether a state of a proclet system is a configuration.

Definition 6 (Configuration). *A* configuration $K = (\mathcal{I}, m_S, m_C)$ *of a proclet system* $\mathcal{P} = (\{P_1, \ldots, P_n\}, C)$ *is defined as follows:*

- *The set $\mathcal{I}$ defines the active proclet instances in the system (as a set of instance ids).*
- *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.*
- *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.*

The initial *configuration $K_0 := (\emptyset, m_{S,0}, m_{C,0})$ defines $m_{S,0}(s) = []$, for all $s \in S$, and $m_{C,0}(c) = []$, for all $c \in C$.*

An execution of $\mathcal{P}$ starts in the initial configuration $K_0 = (\emptyset, m_{S,0}, m_{C,0})$ and occurrences of transitions of $\mathcal{P}$ take the system from configuration to configuration, creating and terminating proclet instances as the execution evolves. Each transition t occurs in a specific proclet instance id, thereby consuming a set of messages received from a multiset *SID* of *sender instances* and producing messages to a multiset RID of *recipient instances*. If t is not attached to an input or output port, then *SID* and/or RID are, respectively, always empty for each occurrence of t.

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 (Valid multiset of senders). *Let $\mathcal{P}$ be a proclet system, $K = (\mathcal{I}, m_S, m_C)$ a configuration of $\mathcal{P}$, $P = (N, ports)$ a proclet of $\mathcal{P}$ 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 any 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 $c = (q, p) \in C$ and the channel contains messages from senders $X = \{id_s \mid (id_s, id_r) \in m_C(c) \wedge id_r = id\}$ then*

1. $card_p = 1$ *implies* $SID \subseteq X$ *and* $|SID| = 1$,
2. $card_p = ?$ *implies* $SID \subseteq X$ *and if* $X \neq \emptyset$ *then* $|SID| = 1$,
3. $card_p = +$ *implies* $SID = X$ *and* $|SID| \geq 1$, *and*
4. $card_p = *$ *implies* $SID = X$.

Validity of the recipient ids RID w.r.t. t is defined correspondingly: RID is either empty (if t has no output port), or RID satisfies the cardinality constraint of its output port (i.e., 1 implies $|RID| = 1$, ? implies $|RID| \in \{0, 1\}|$, + implies $|RID| \geq 1$).

Definition 8 (Enabled transition). *Let $\mathcal{P}$ be a proclet system, $K = (\mathcal{I}, m_S, m_C)$ a configuration of $\mathcal{P}$, P a proclet of $\mathcal{P}$. A transition $t \in T_P$ is* enabled *in instance id at configuration K w.r.t. multisets SID of sender- and RID of recipient ids, iff*

1. *if t has no pre-place, then $id \notin \mathcal{I}$ (an initial transition creates a new instance),*
2. *each pre-place s of t in instance id has a token, i.e., $m_S(s)(id) > 0$, and*
3. *SID and RID are both valid with respect to t.*

When an enabled transition t *occurs*, it takes the system from a configuration $K = (\mathcal{I}, m_S, m_C)$ to the *successor configuration* $K' = (\mathcal{I}', m'_S, m'_C)$, depending on SID and RID.

Definition 9 (Occurrence of a transition). *Let $\mathcal{P}$ be a proclet system, K a configuration of $\mathcal{P}$, and P a proclet of $\mathcal{P}$. If transition $t \in T_p$ is enabled in instance id at configuration K w.r.t. SID and RID, then t can* occur *which defines an instance-aware event $e = (\ell_{\mathcal{P}}(t), id, SID, RID)$, and yields the successor configuration K' as follows:*

1. *in instance id, consume a token from each pre-place s of t, and produce a token on each post-place s of t,*
2. *if t is attached to an input port p, then for each $id_s \in SID$, consume from channel $c = (q, p) \in C$ as many messages (id_s, id) as expected, i.e., $SID(id_s)$ messages, and*
3. *if t is attached to an output port p, then for each $id_r \in RID$, produce on channel $c = (p, q) \in C$ as many messages (id, id_r) as intended, i.e., $RID(id_r)$ messages.*

An execution of the proclet system $\mathcal{P}$ is a sequence $K_0 \xrightarrow{e_1} K_1 \xrightarrow{e_2} \dots K_n$ where each K_{i+1} is the successor configuration of K_i under the instance-aware event e_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, 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. 1.

Note that proclets may have τ-labeled transitions which are usually interpreted as internal or *unobservable* transitions. The corresponding instance-aware event $e = (\tau, id, SID, RID)$ *would not be recorded in an event log*. Replaying a log on a model with unobservable transitions is the main technical problem addressed by conformance checking as we discuss next.

3 The Interaction Conformance Problem

The problem of determining how accurately a process model describes the process implemented in an actual information system S is called *conformance checking problem* [2].

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 process model M define the set of valid process executions as sequences of M's actions. Conformance of M to L can be characterized in several dimensions [2]. 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., [7]. 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.

There exist several techniques for the classical conformance checking problem [2–7, 14, 15]. The approaches compare cases in a log with possible executions of a process model, often by replaying the log on the model to see where model and log deviate. The most advanced conformance metrics reflect that only parts of a case are deviating [14], and pinpoint where deviations occur [6], while taking into account that models may contain behavior that is unobservable by nature [7, 15]. The latter techniques 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. This particulary allows to extend σ' with τ-labeled transitions not recorded in σ, so σ' can be replayed on a model with unobservable transitions (see Sect. 2.4).

A proclet system raises a more general conformance checking problem, because a case contains events of several proclet instances 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 $\mathcal{P}$, it 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. In typical case studies we found logs with 80,000 events of 40-60 distinct actions. Checking conformance would define a search space of $60^{80,000}$ possible solutions among which the most similar log $\mathcal{L}'$ has to be found. Even exploring only a small fraction of such a search space quickly turns out infeasible. Moreover, existing techniques would be unable to distinguish the difference instances. For these two reasons, we decompose the problem and reduce it to a classical conformance checking problem. Here we will use the technique presented in [7, 15] which is most robust and flexible.

4.1 Reducing Artifact Conformance to Existing Techniques

A naive solution would be to simply decompose the conformance problem of proclet system $(\{P_1, \ldots, P_n\}, C)$ and instance-aware event log $\mathcal{L} = (\{L_1, \ldots, L_n\}, <)$ into n smaller problems where classical techniques are used to compare P_i and L_i. However, it is not sufficient as the life-cycle of some case id does not only depend on "local" events, but also on events that sent messages to id or received messages from id. So, all events of id *together* with all events of $\mathcal{L}$ that exchange messages with id constitute the *interaction case* $\overline{\sigma^{id}}$ of id. It contains all behavioral information showing how 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 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 $\mathcal{P} = P_1 \oplus \ldots \oplus P_n$ 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 $P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n$.

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

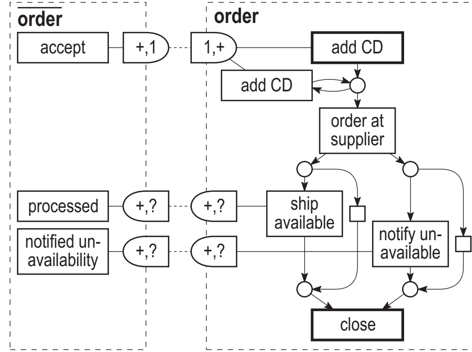

Fig. 3. The proclet order of Fig. 1 together with its environment $\overline{\text{order}}$.

Figure 3 shows the proclet order together with its abstracted environment $\overline{\text{order}}$ from the proclet system of Fig. 1. The formal definition reads as follows.

Definition 10 (Environment Abstraction). *Let $\mathcal{P} = (\{P_1, \ldots, P_n\}, C)$ be a proclet system, $P_i = (N_i, ports_i), N_i = (S_i, T_i, F_i, \ell_i), i = 1, \ldots, n$ with the global labeling*

$\ell_{\mathcal{P}}(t) = \ell_i(t), t \in T_i, i = 1, \ldots, n$. *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* $\overline{P_i} = (N, ports)$ *with* $N = (\emptyset, \overline{T_i}, \emptyset, \ell_{\mathcal{P}}|_{\overline{T_i}})$, *and* $ports = \{q \mid (p,q) \in C_i \cup C_i^{-1}, q \notin ports_i\}$. *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

After decomposing the proclet system, the next step is to check conformance of each single proclet P_i with 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 exchange messages with the instance id of this case.

Definition 11 (Interaction case, interaction log). *Let* $\mathcal{L} = (\{L_1, \ldots, L_n\}, <)$ *be an instance-aware log. Let* $\mathcal{E}$ *be the set of all events in all cases in* $\mathcal{L}$. *Let* P_i *be a proclet of a proclet system* $\mathcal{P} = P_1 \oplus \ldots \oplus P_n$, $i \in \{1, \ldots, n\}$. *Let* $\sigma \in L_i$ *be a case of an instance* id *of* P_i.

The set $\mathcal{E}|_{id}$ *of events of* $\mathcal{L}$ *that involve* id *contains event* $e = (a, id', SID, RID)$ *iff* $e \in \mathcal{E} \wedge (id' = id \vee id \in SID \vee id \in RID)$. *The* interaction case *of* σ *is the sequence* $\overline{\sigma}$ *containing all events* $\mathcal{E}|_{id}$ *ordered by* $<$ *of* $\mathcal{L}$. *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} shown in Section 2.2 are

$\overline{\sigma_{o1}}$: $\langle(\mathsf{accept}, q_1, [], [o_1]), (\mathsf{accept}, q_2, [], [o_1, o_2, o_2]), (\mathsf{add\ CD}, o_1, [q_1], []),$
$(\mathsf{add\ CD}, o_1, [q_2], []), (\mathsf{order}, o_1, [], []), (\mathsf{ship}, o_1, [], [q_1, q_2]), (\mathsf{processed}, q_1, [o_1], []),$
$(\mathsf{processed}, q_2, [o_1, o_2], []), (\mathsf{close}, o_1, [], [])\rangle$

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

The abstracted proclet system $\mathsf{quote} \oplus \overline{\mathsf{quote}}$ can replay both interaction cases.

4.4 The Decomposition is Correct

Decomposing a proclet system $\mathcal{P} = P_1 \oplus \ldots \oplus P_n$ 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 $i = 1, \ldots, n$ equivalently preserves the fitness of $\mathcal{P}$ w.r.t. the given instance-aware event log $\mathcal{L}$.

Recall from Sect. 2.4 that $\mathcal{L}$ is replayed on $\mathcal{P}$ by ordering all events of $\mathcal{L}$ in a single case σ. From Def. 11 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. By induction then holds that $P_1 \oplus \ldots \oplus P_n$ can replay σ iff each proclet with its environment $P_i \oplus \overline{P_i}$ can replay each projection of $\overline{\sigma^{id}}$ onto events of each instance id of P_i. In other words, $P_1 \oplus \ldots \oplus P_n$ fits $\mathcal{L}$ iff $P_i \oplus \overline{P_i}$ fits each interaction case of P_i, i.e. $P_i \oplus \overline{P_i}$ fits $\mathcal{L}|_{P_i}$. The full proof is given in [16].

4.5 Checking Interaction Conformance

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 $P_i \oplus \overline{P_i}$ can replay the interaction log $\mathcal{L}|_{P_i}$, where each case in $\mathcal{L}|_{P_i}$ only refers to exactly one proclet instance. Thus, the problem can be fed into existing conformance checkers.

Our conformance checker leverages the technique described in [7, 15]. As this technique only takes Petri nets as input, the conformance checking problem of $P_i \oplus \overline{P_i}$ w.r.t. $\mathcal{L}|_{P_i}$ is further reduced to a conformance checking problem on Petri nets. This 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 $\mathcal{L}|_P$. Fig. 4 shows an example. Each channel of Fig. 3 translates to a place, the port annotations of proclet order translate to additional nodes and arcs.

For instance, cardinality $+$ of the output port of ship available translates to the Petri net pattern highlighted grey in Fig. 4. It ensures that an occurrence of ship available yields one (t_2) or more messages (t_1) on the channel. The multiplicity $+$ of the input port of add CD translates to place p_2 and adjacent arcs: each occurrence of add CD produces a token, the normal arc to the final transition close ensures that add CD occurred at least once during the lifetime of order, the *reset arc* (double arrow head) removes all other tokens on p_2 (allowing for multiple occurrences of add CD), the *inhibitor arc* from the channel to close ensures that all messages were consumed; see [16] for details.

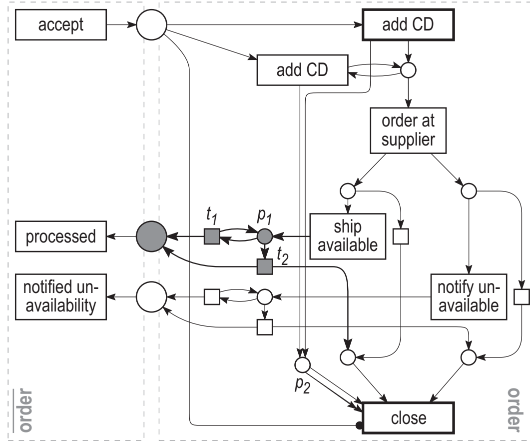

Fig. 4. The result of translating order $\oplus$ $\overline{\text{order}}$ of Fig. 3 to a Petri net N_{order}.

Moreover, the technique described in [7,15] is not aware of interaction cases (Def. 11). In particular, it is unaware that event $e = (a, id', [id, id, id''], [])$ sends two messages to instance id and one message to instance id''. In our translation of ports to Petri nets [16], each event e may produce one message to one instance. In order to preserve the number of messages sent to instance id in an interaction case, we replace e by as many occurrences of a as e sends messages to id, i.e., two occurrences of a. For example, the interaction case $\overline{\sigma_{o2}}$ (Sect. 4.3) is transformed to $\langle$accept, accept, add CD, add CD, order, notify, notified, ship, processed, close$\rangle$, which can be replayed on the net of Fig. 4. Further details are given in [16].

After converting the proclets into Petri nets $N_{P_1}, \ldots, N_{P_n}$ and translating their interaction cases as mentioned above, our conformance checker applies the technique

of [7, 15] to check how good the net N_{P_i} replays $\mathcal{L}|_{P_i}$, for each $i = 1, \ldots, n$ 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 [7, 15]; this technique also checks conformance of cyclic models. 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 $P_1 \oplus \ldots \oplus P_n$ w.r.t. $\mathcal{L}$ is currently computed as the average fitness of each P_i to its interaction cases $\mathcal{L}|_{P_i}$. Alternatively, a *weighted* average could measure the fitness of the entire proclet system: a proclet's weight could be for instance its size (i.e., number of transitions), or the size of its interface (i.e., the number of ports, measuring the amount of interaction of the proclet).

To illustrate the misconformances that can be discovered with this technique, assume that in the process execution of Fig. 2, case q_1 did not contain an accept event. This would lead to the following interaction case of o_1:

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

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.

5 Implementation as a ProM plug-in

The *interaction conformance checker* is implemented as a software plug-in of ProM, a generic open-source framework for implementing process mining tools in a standard environment [17]. The plug-in takes as input the proclet system model and the interaction logs of the proclet of interests and, by employing the techniques described in Sect. 4, returns an overview of the deviations between the cases in the log and the proclet system model.

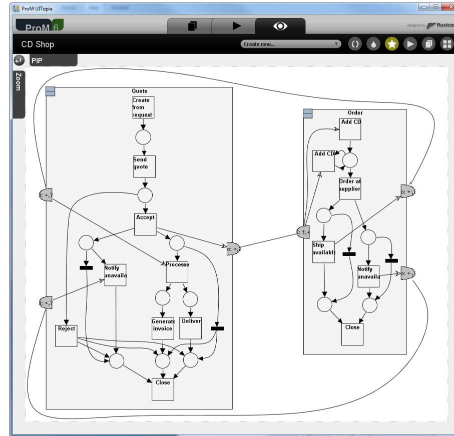

Fig. 5. The CD shop example in ProM.

As input for initial experiments, we generated event logs by modeling our CD shop example as an artifact-centric system in CPN Tools (http://cpntools.org) and simulating the model. The simulation yielded to instance-aware log of about 2914 events of 15 different types. Next we created the

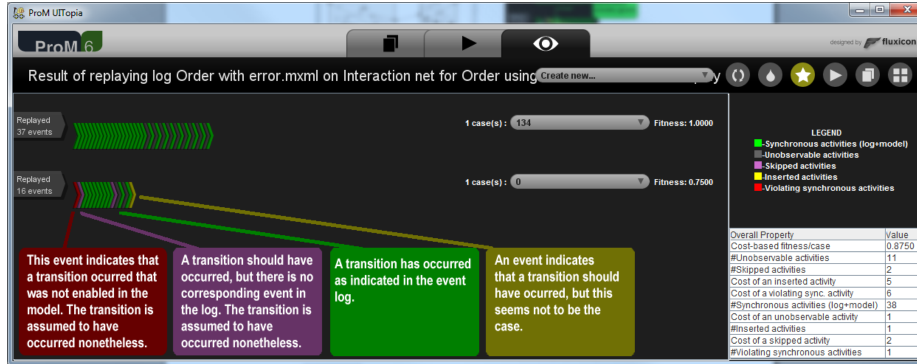

Fig. 6. The conformance results for the *order* proclet in ProM.

proclet model of the CD shop which is visualized in ProM as shown in Fig. 5; invisible transitions are depicted as black rectangles. Then we generated interaction logs from the instance-aware log (Def. 11). The longest interaction case was part of the interaction log of quote and contained 31 events over 11 types. Therefore, our approach of decomposing the artifact conformance-checking problem into a set of smaller sub-problems reduced the worst-case search space size from 15^{2914} to 11^{31}.

For conformance checking, we implemented generic conversions from proclet systems to Petri nets as explained in Sects. 4.2 and 4.5. The resulting Petri nets were then checked for conformance w.r.t. the respective interaction logs using the existing conformance checker [15], capable to replay logs on Petri Nets with reset- and inhibitor arcs.

The result of the conformance checking is shown in Figure 6. 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 replayed execution 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 the 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 has fitness 1. Note that the conformance checker identified some invisible transitions to have fired (indicated by the black triangles). These are the transitions necessary in the Petri net to model cardinality of the ports. The second case shows a lower conformance. The conformance checker identifies where the case and the model deviate; and a color coding indicates the type of deviation.

6 Conclusion

In this paper, we considered the problem of determining whether different objects of a process interact according to their specification. We took 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 to model many-to-many relationships and complex interactions between objects.

Existing conformance checking techniques allow only for checking the conformance of artifacts in isolation. In this paper we went beyond and checked whether the complex

interactions among artifacts that are proposed in an artifact model fit the actual behavior observed in reality. In particular, we showed that the problem of interaction conformance can be decomposed a set of smaller sub-problems for which we can use classical conformance checking techniques. The feasibility of the approach is demonstrated by a concrete operationalization in the ProM framework.

An open problem is to generalize instance-aware events to have correlation data defined by multiple correlation properties, and correspondingly to allow proclet transitions to be attached to an arbitrary number of ports.

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. Rozinat, A., Jong, I., Gunther, C., van der Aalst, W.: Conformance Analysis of ASML's Test Process. In: GRCIS'09. Volume 459 of CEUR-WS.org. (2009) 1–15
2. Rozinat, A., de Medeiros, A.K.A., Günther, C.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The Need for a Process Mining Evaluation Framework in Research and Practice. In: BPM'07 Workshops. Volume 4928 of LNCS., Springer (2007) 84–89
3. Greco, G., Guzzo, A., Pontieri, L., Sacca, D.: Discovering Expressive Process Models by Clustering Log Traces. IEEE Trans. on Knowl. and Data Eng. **18** (2006) 1010–1027
4. Weijters, A., van der Aalst, W.: Rediscovering Workflow Models from Event-Based Data using Little Thumb. Integrated Computer-Aided Engineering **10** (2003) 151–162
5. Medeiros, A., Weijters, A., van der Aalst, W.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery **14** (2007) 245–304
6. Rozinat, A., van der Aalst, W.: Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems **33** (2008) 64–95
7. Adriansyah, A., van Dongen, B., van der Aalst, W.: Towards Robust Conformance Checking. In: Business Process Management Workshops. Volume 66 of LNBIP. (2010) 122–133
8. 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
9. Nigam, A., Caswell, N.: Business artifacts: An Approach to Operational Specification. IBM Systems Journal **42** (2003) 428–445
10. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. **32** (2009) 3–9
11. 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
12. Lohmann, N., Wolf, K.: Artifact-centric choreographies. In: ICSOC 2010. Volume 6470 of LNCS., Springer (2010) 32–46
13. Barros, A.P., Decker, G., Dumas, M., Weber, F.: Correlation patterns in service-oriented architectures. In: FASE. Volume 4422 of LNCS., Springer (2007) 245–259
14. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
15. Adriansyah, A., Sidorova, N., van Dongen, B.: Cost-based fitness in conformance checking. In: ACSD'11. (2011) To appear.
16. Fahland, D., de Leoni, M., van Dongen, B., van der Aalst, W.: Checking behavioral conformance of artifacts. BPM Center Report BPM-11-08, BPMcenter.org (2011)
17. Verbeek, H., Buijs, J.C., van Dongen, B.F., van der Aalst, W.M.P.: ProM: The Process Mining Toolkit. In: BPM Demos 2010. Volume 615 of CEUR-WS. (2010)