Conformance Checking Using Object-Centric Behavioral Constraints

Wil M.P. van der Aalst[1], Guangming Li[1], and Marco Montali[2]

[1] Eindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tue.nl, g.li.3@tue.nl
[2] Free University of Bozen-Bolzano, Piazza Domenicani 3, I-39100, Bolzano, Italy.
montali@inf.unibz.it

Abstract. Today's process modeling languages often force the analyst or modeler to straightjacket real-life processes into simplistic or incomplete models that fail to capture the essential features of the domain under study. Conventional business process models only describe the lifecycles of individual instances (cases) in isolation. Although process models may include data elements (cf. BPMN), explicit connections to *real* data models (e.g. an entity relationship model or a UML class model) are rarely made. Therefore, we propose a novel notation that *extends data models with a behavioral perspective*. Data models can easily deal with many-to-many and one-to-many relationships. This is exploited to create process models that can also model complex interactions between different types of instances. Classical multiple-instance problems are circumvented by using the data model for event correlation. The declarative nature of the proposed language makes it possible to model behavioral constraints over activities like cardinality constraints in data models. The resulting *object-centric behavioral constraint model* is able to describe processes involving *interacting instances* and *complex data dependencies*. This model can be then used for *conformance checking*, i.e., diagnosing discrepancies between actual behavior and modeled behavior. We have developed several *ProM* plug-ins to edit and check object-centric behavioral constraint models. Experiments show that we can now detect and diagnose a range of conformance problems that would have remained undetected using conventional process-model notations. In particular, we illustrate the usefulness of this new approach by extracting data from the ERP/CRM system *Dolibarr*.

1 Introduction

Techniques for business process modeling (e.g., BPMN diagrams, Workflow nets, EPCs, or UML activity diagrams) tend to suffer from two main problems:

- It is *difficult to model interactions between process instances*, which are in fact typically considered in isolation. Concepts like lanes, pools, and message flows in conventional languages like BPMN aim to address this. However, within each (sub)process still a single instance is modeled in isolation.
- It is also *difficult to model the data-perspective and control-flow perspective in a unified and integrated manner*. Data objects can be modeled, but the more powerful constructs present in Entity Relationship (ER) models and UML class models cannot be expressed well in process models. For example, cardinality constraints

in the data model *must* influence behavior, but this is not reflected at all in today's process models.

Because of these problems there is a mismatch between process models and the data in (and functionality supported by) real enterprise systems from vendors such as SAP (S/4HANA), Microsoft (Dynamics 365), Oracle (E-Business Suite), and Salesforce (CRM). These systems are also known as Enterprise Resource Planning (ERP) and/or Customer Relationship Management (CRM) systems and support business functions related to sales, procurement, production, accounting, etc. These systems may contain hundreds, if not thousands, of tables with information about customers, orders, deliveries, etc. For example, SAP has tens of thousands of tables. Also Hospital Information Systems (HIS) and Product Lifecycle Management (PLM) systems have information about many different entities scattered over a surprising number of database tables. Even though a clear process instance notion is missing in such systems, mainstream business process modeling notations can only describe the lifecycle of one type of process instance at a time. The disconnect between process models and the actual processes and systems becomes clear when applying process mining using data from enterprise systems. How to discover process models or check conformance if there is no single process instance notion?

The problems mentioned have been around for quite some time (see for example [**?**]), but were never solved satisfactorily. Artifact-centric approaches [**?**,**?**,**?**,**?**] (including the earlier work on proclets [**?**]) attempt to address the above problems. However, these approaches tend to result in models where

- the description of the end-to-end behavior needs to be distributed over multiple diagrams (e.g., one process model per artifact),
- the control-flow cannot be related to an overall data model (i.e., there is no explicit data model or it is separated from the control-flow), and
- interactions between different entities are not visible or separated (because artifacts are distributed over multiple diagrams).

Within an artifact, proclet, or subprocess, *one is forced to pick a single instance notion*. Moreover, cardinality constraints in the data model cannot be exploited while specifying the intended dynamic behavior. We believe that data and process perspectives can be unified better, as demonstrated in this paper.

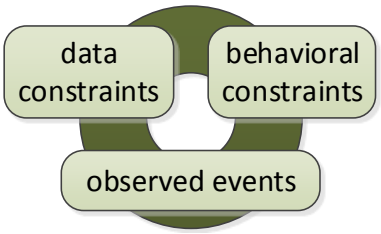

Fig. 1. Object-Centric Behavioral Constraint (OCBC) models connect data constraints (like in a UML class diagram), behavioral constraints (like in a process model or rule set), and real event data. This allows for novel forms of conformance checking.

This paper proposes the *Object-Centric Behavioral Constraint* (OCBC) model as a novel language that combines ideas from declarative, constraint-based languages like *Declare* [**?**], and from data/object modeling techniques (ER, UML, or ORM). Cardi-

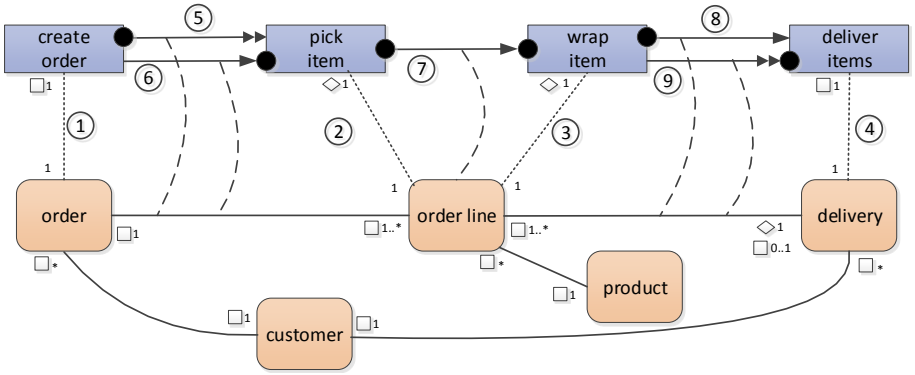

Fig. 2. A small *Object-Centric Behavioral Constraint* (OCBC) model.

nality constrains are used as a *unifying mechanism* to tackle data and behavioral dependencies, as well as their interplay (cf. Figure 1). The primary application considered in this paper is *conformance checking* [**?**,**?**,**?**,**?**]. Deviations between observed behavior (i.e., an event log) and modeled behavior (OCBC model) are diagnosed for compliance, auditing, or risk analysis. Unlike existing approaches, instances are not considered in isolation and cardinality constraints in the data/object model are taken into account. Hence, problems that would have remained undetected earlier, can now be detected.

Figure 2 shows an OCBC model with four activities (*create order*, *pick item*, *wrap item*, and *deliver items*) and five object classes (*order*, *order line*, *delivery*, *product*, and *customer*). The top part describes the ordering of activities and the bottom part the structuring of objects relevant for the process. The lower part can be read as if it was a UML class diagram. Some cardinality constraints should hold at any point in time as indicated by the $\Box$ ("always") symbol. Other cardinality constraints should hold from some point onwards as indicated by the $\Diamond$ ("eventually") symbol. Consider for example the relation between *order line* and *delivery*. At any point in time a delivery corresponds to one of more order lines (denoted $\Box$ $1..^*$) and an order line refers to as most one delivery (denoted $\Box$ $0..1$). However, eventually an order line should refer to precisely one delivery (denoted $\Diamond$ 1). Always, an order has one or more order lines, each order line corresponds to precisely one order, each order line refers to one product, each order refers to one customer, etc. The top part shows behavioral constraints and the middle part relates activities, constraints, and classes.

The notation used in Figure 2 will be explained in more detail later. However, to introduce the main concepts, we first informally describe the 9 constructs highlighted in Figure 2. (1) There is a one-to-one correspondence between *order* objects and *create order* activities. If an object is added to the class *order*, the corresponding activity needs to be executed and vice versa. (2) There is a one-to-one correspondence between *order line* objects and *pick item* activities. (3) There is also a bijection between *order line* objects and *wrap item* activities. The $\Diamond$ 1 annotations next to *pick item* and *wrap item* indicate that these activities need to be executed for every order line. However,

they may be executed some time after the order order line is created. ④ There is a one-to-one correspondence between *delivery* objects and *delivery items* activities. ⑤ Each *create order* activity is followed by one or more *pick item* activities related to the same order. ⑥ Each *pick item* activity is preceded by precisely one corresponding *create order* activity. ⑦ Each *pick item* activity is followed by one *wrap item* activity corresponding to the same order line. Each *wrap item* activity is preceded by one *pick item* activity corresponding to the same order line. ⑧ Each *wrap item* activity is followed by precisely one corresponding *deliver items* activity. ⑨ Each *deliver items* activity is preceded by at least one corresponding *wrap item* activity. A *deliver items* activity is implicitly related to a set of order lines through the relationship between class *order line* and class *delivery*. The notation will be explained later, however, note that a single order may have many order lines that are scattered over multiple deliveries. Moreover, one delivery may combine items from multiple orders for the same customer.

The process described in Figure 2 cannot be modeled using conventional notations (e.g., BPMN) because (a) three different types of instances are intertwined and (b) constraints in the class model influence the allowed behavior. Moreover, the OCBC model provides a *full* specification of the allowed behavior in a *single diagram*, so that no further coding or annotation is needed.

To support OCBC models several ProM plug-ins have been developed: an OCBC model editor and viewer, a dedicated log viewer, and a conformance checker taking an OCBC model and event log as input.[3] The conformance checker summarizes the deviations found and highlights conformance problems both in the model and log.

Because OCBC models do not assume a single instance notion and data and control-flow are tightly coupled (e.g., relations in the object model influence obligations), we cannot use conventional event logs (e.g., in XES or MXML format). Instead, the OCBC plug-ins rely on so-called XOC logs, i.e., event logs without an instance notion but with events referring to objects in an evolving object model. Such logs can be extracted from any information system centered around a database (e.g., ERP, CRM, HIS, and PLM systems). To demonstrate this we extract XOC logs from the open-source ERP/CRM system *Dolibarr*. The same approach can be used to extract XOC logs from SAP or Oracle.

The remainder is organized as follows. Related work is discussed in Section 2. Section 3 shows that control-flow can be modeled as cardinality constraints over sets of predecessors and successors. Section 4 defines class models and object models. Both views are integrated in Section 5 where OCBC models are proposed. Section 6 provides the semantics for OCBC models and demonstrates that novel forms of conformance checking are possible. OCBC models the corresponding conformance checking approach are supported the ProM plug-ins described in Section 7. Section 8 describes a case study showing that we can XOC logs from ERP/CRM systems and detect conformance problems that would remain undetected using conventional approaches. Section 9 concludes the paper.

[3] Download ProM from `promtools.org` and install the *OCBC package*.

2 Related Work

To position the work in literature on business process modeling and process mining, it is important start from the two key problems faced by mainstream process modeling notations:

- It is difficult to model interactions between process instances.
- It is difficult to model the data-perspective and control-flow perspective in a unified and integrated manner.

OCBC models aim to address these problems because they are essential when relating process models to the actual data in existing real-life enterprise systems. After relating OCBC models to alternative process modeling notations (Section 2.1) and data modeling approaches supporting temporal constraints (Section 2.2), we discuss related techniques in process mining Section 2.3.

2.1 Process Modeling Notations Capturing Data and Multiple Instances

Over time there have been numerous attempts to add data to process models. Consider for example the various types of *colored Petri nets*, i.e., Petri nets where tokens have a value [**?**,**?**,**?**,**?**,**?**]. These approaches do not support explicit data modeling as can be found in Entity-Relationship (ER) models [**?**], UML class models [**?**], and Object-Role Models (ORM) [**?**]. Places and tokens are often typed but there is no data model to relate entities and activities.

The first approaches that explicitly related process models and data models appeared in the 1990-ties [**?**,**?**]. See for example the approach by Kees van Hee [**?**] who combined (1) Petri nets, extended with time, token values and hierarchy, (2) a specification language that is a subset of Z, and (3) a binary data model, extended with complex objects.

Object-aware process management, as supported by the *PHILharmonicFlows* framework [**?**,**?**], provides an integrated methodology for modeling requirements of process- and data-centric software systems. Key elements of the approach are the separation of concerns (data and process can be considered separately) and the ability to support change.

Artifact-centric approaches [**?**,**?**,**?**,**?**] (including the earlier work on proclets [**?**]) aim to capture business processes in terms of so-called business artifacts, i.e., key entities driving a company's operations and whose lifecycles and interactions define an overall business process. Artifacts have data and lifecycles attached to them, thus relating both perspectives.

Other approaches integrating data and processes include *case handling systems* [**?**] and *product-based workflows* [**?**]. In fact, it is impossible to list all the different proposals described in literature.

The approach in this paper is different from the above approaches, because the data-perspective and control-flow perspective are modeled in a single diagram and while using a single unifying mechanism, namely *cardinality constraints*.

The control-flow part of OCBC models is inspired by *Declare* [**?**]. In fact, OCBC models can be seen as declarative models. Declarative process models like *Declare* have been formalized in terms of Linear Temporal Logic (LTL) over finite traces [**?**], using abductive logic programming [**?**], Event Calculus [**?**], regular expressions [**?**], and

colored automata [**?**]. However, none of these declarative approaches allows for data modeling (other than introducing variables and guards as done in the *Declare* workflow system).

2.2 Data Modeling Approaches

⟨TODO: Input Marco. Also adapt title.⟩

2.3 Data-Aware Process Mining Techniques

The idea to use OCBC models emerged from problems encountered when applying *process mining* to data in enterprise systems. The many tables in ERP, CRM, HIS, and PLM systems illustrate that it is rather naive to assume a single instance notion when analyzing a process (see Section 1). Automatically discovering process models for a specific type of instance often leads to small, disconnected, and trivial models. *Interactions between different instance types and between control-flow and data are key for process mining.*

Process mining [**?**] is a process centric technique that helps to turn event data into real value: by discovering the real processes, by automatically identifying bottlenecks, by analyzing deviations and sources of non-compliance, by revealing the actual behavior of people, etc. Although the field is broader, three main types of process mining can be identified *process discovery*, *conformance checking*, and *performance analysis*.

A process discovery technique takes an event log as an input and produces a model without using any a-priori information. Typically the focus of process discovery techniques in on the control-flow aspect of a process. Examples are the α-algorithm [**?**], heuristic mining [**?**], fuzzy mining [**?**], region-based techniques [**?**,**?**,**?**], and inductive mining techniques [**?**,**?**].

For conformance checking an existing process model is compared with an event log of the process that the model is describing, i.e., modeled behavior is confronted with observed behavior [**?**,**?**]. Conformance checking can be used to check if reality, as recorded in the log, conforms to the model and vice versa. Most conformance checking techniques are based on replaying the traces in the event log on the model. In [**?**] the number of missing and remaining tokens are counted while replaying the event log. If an activity in the event log is not enabled, then a missing token is added and at the end the non-used tokens are counted. This can used to diagnose control-flow problems. State-of-the-art in conformance checking are the so-called alignment-based approaches [**?**,**?**]. Given a trace in the event log, a closest path in the model is computed by solving an optimization problem. Most techniques focus on fitness, however, there are also techniques for computing other quality dimensions such as simplicity, precision, and generalization.

Techniques for conformance checking can be combined with timing information present in the event log to analyze performance [**?**]. For example, by combining alignments and timestamps in the event log one can deduce the average time in-between two activities.

Performance analysis obviously focusses on the time perspective. Most process discovery and conformance checking focus exclusively on control-flow. However, there are also a few process mining approaches that discover or check process models with data. See for example [**?**,**?**] for conformance checking of Petri nets extended with data.

One can also apply standard data mining techniques to decision points in processes to learn guards [**?**].

A few discovery techniques have been developed for artifact-centric process models [**?**,**?**]. The approach in [**?**] starts from a raw event stream and learns correlation information between the events to build the an Entity-Relationship (ER) model. Guided by the user a so-called *artifact-centric log* is created. Such artifact-centric logs are used to discover the lifecycles of artifacts. Here any approach can be used that produces a Petri net. In the final step these are translated into the Guard-Stage-Milestone (GSM) notation [**?**]. The approach in [**?**] uses a different starting point: relational transactional data from an ERP system is exploited to learn a process model describing all transactions and their order. The approach can detect deviations and deal with *convergence* (one event is related to multiple cases) and *divergence* (a case is related to multiple events of the same event type) [**?**]. The result is a set of interconnected lifecycle models. The technique may discover interactions between artifacts at the type level and the event level. Unlike OCBC models, the different instance notions are separated in different diagrams and the result does not show the interplay between data and control-flow as shown in Figure 2.

Although OCBC models can be used for all three main types of process mining, we focus on *conformance checking*. Most related are [**?**,**?**] that check the conformance of artifact-centric models expressed in terms of proclets [**?**]. These papers show that process instances *cannot* be considered in isolation as instances in artifact-centric processes may overlap and interact with each other. This complicates conformance checking but the problem can be decomposed into a set of smaller problems that can be analyzed using conventional conformance checking techniques [**?**,**?**]. These artifact-centric conformance checking techniques do not relate control-flow to some overall data model (like the class model in OCBC models).

Also related is the work on conformance checking of *declarative models* [**?**]. However, this work does not consider multiple intertwined instance notions and also does not relate control-flow to some overall data model as in Figure 2.

Moreover, none of the approaches mentioned uses cardinality constrains as a *unifying mechanism* to tackle data and behavioral dependencies, as well as their interplay.

3 Modeling Behavioral Cardinality Constraints

In this paper, a process is merely a collection of *events* without assuming some case or process instance notion. Each event corresponds to an *activity* and may have additional *attributes* such as the time at which the event took place, the resource executing the corresponding event, the type of event (e.g., start, complete, schedule, abort), the location of the event, or the costs of an event. Each event attribute has a *name* (e.g., "age") and a *value* (e.g., "49 years"). Moreover, events are atomic and ordered. For simplicity, we assume a *total order*. To model the overlapping of activities in time one can use start and complete events.

Definition 1 (Events and Activities). $\mathcal{U}_E$ *is the universe of* events*, i.e., things that happened.* $\mathcal{U}_A$ *is the universe of* activities*.*

- *Function $act \in \mathcal{U}_E \rightarrow \mathcal{U}_A$ maps events onto activities.*
- *Events can also have additional attributes (e.g., timestamp, cost, resource, or amount). $\mathcal{U}_{Attr}$ is the universe of* attribute names. *$\mathcal{U}_{Val}$ is the universe of* attribute values. *$attr \in \mathcal{U}_E \rightarrow (\mathcal{U}_{Attr} \not\rightarrow \mathcal{U}_{Val})$ maps events onto a partial function assigning values to some attributes,*[4]
- *Relation $\preceq \subseteq \mathcal{U}_E \times \mathcal{U}_E$ defines a total order on events.*[5]

Unlike traditional event logs [**?**] we do *not* assume an explicit case notion. Normally, each event corresponds to precisely one case, i.e., a process instance. In the *Object-Centric Behavioral Constraint models* (OCBC models) described in Section 5 we do not make this restriction and can express *complex interactions between a variety of objects in a single diagram*. However, to gently introduce the concepts, we first define *constraints over a collection of ordered events*.

Definition 2 (Event Notations). *Let $E \subseteq \mathcal{U}_E$ be a set of events ordered by $\preceq$ and related to activities through function act. For any event $e \in E$:*

- *$\trianglelefteq_e(E) = \{e' \in E \mid e' \preceq e\}$ are the events* before *and* including *e.*
- *$\trianglerighteq_e(E) = \{e' \in E \mid e \preceq e'\}$ are the events* after *and* including *e.*
- *$\triangleleft_e(E) = \{e' \in E \mid e' \prec e\}$ are the events* before *e.*[6]
- *$\triangleright_e(E) = \{e' \in E \mid e \prec e'\}$ are the events* after *e.*
- *$\partial_a(E) = \{e' \in E \mid act(e') = a\}$ are the events corresponding to activity $a \in \mathcal{U}_A$.*

A process model can be viewed as a set of *constraints*. In a procedural language like Petri nets, places correspond to constraints: removing a place may allow for more behavior and adding a place can only restrict behavior. In this paper, we will employ a graphical notation inspired by *Declare* [**?**] (see Section 2). Here, we provide a formalization of a subset of *Declare* in terms of *behavioral cardinality constraints*. This allows us to reason about behavior and data in a *unified* manner, since both use cardinality constraints. The following cardinality notion will be used to constrain both data and behavior.

Definition 3 (Cardinalities). *$\mathcal{U}_{Card} = \{X \subseteq \mathbb{N} \mid X \neq \emptyset\}$ defines the universe of all possible cardinalities. Elements of $\mathcal{U}_{Card}$ specify non-empty sets of integers.*

Cardinalities are often used in data modeling, e.g., Entity-Relationship (ER) models and UML Class models may include cardinality constraints. Table 1 lists a few shorthands typically used in such diagrams. For example, "1..*" denotes any positive integer.

In line with literature, we adopt the notation in Table 1 for cardinality constraints over data. For behavioral cardinality constraints, we adopt a different notation, but very similar in spirit. Given some *reference event* e we can reason about the events *before* e and the events *after* e. We may require that the cardinality of the set of events corresponding to a particular activity before or after the reference event lies within a particular range.

[4] $f \in X \not\rightarrow Y$ is a partial function with domain $dom(f) \subseteq X$.

[5] A total order is a binary relation that is (1) antisymmetric, i.e. $e_1 \preceq e_2$ and $e_2 \preceq e_1$ implies $e_1 = e_2$, (2) transitive, i.e. $e_1 \preceq e_2$ and $e_2 \preceq e_3$ implies $e_1 \preceq e_3$, and (3) total, i.e., $e_1 \preceq e_2$ or $e_2 \preceq e_1$.

[6] $e' \prec e$ if and only if $e' \preceq e$ and $e' \neq e$.

notation	allowed cardinalities
1	$\{1\}$
$1..k$	$\{1, 2, \ldots, k\}$
$*$	$\{0, 1, 2, \ldots\}$
$1..*$	$\{1, 2, \ldots\}$

Table 1. Some examples of frequently used shorthands for elements of $\mathcal{U}_{Card}$.

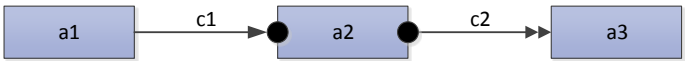

Fig. 3. Two behavioral cardinality constraints: constraint $c1$ specifies that all $a2$ events should be preceded by precisely one $a1$ event and constraint $c2$ specifies that all $a2$ events should be succeeded by at least one $a3$ event.

Consider for example the two constraints depicted in Figure 3. Assume a set of events $E \subseteq \mathcal{U}_E$. The *reference events* for $c1$ are all $a2$ events, i.e., all events $E^{c1}_{ref} = \partial_{a2}(E)$. This is indicated by the black dot connecting the $c1$ arrow to activity $a2$. Now consider a reference event $e_{ref} \in E^{c1}_{ref}$. The single-headed arrow towards the black dot indicates that e_{ref} should be preceded by precisely one $a1$ event. The $a1$ events are called *target events* (counterpart of e_{ref} when evaluating the constraint). Formally, constraint $c1$ demands that $|\lhd_{e_{ref}}(\partial_{a1}(E))| = 1$, i.e., there has to be precisely one $a1$ event before e_{ref}.

The reference events for $c2$ are also all $a2$ events, i.e., $E^{c2}_{ref} = \partial_{a2}(E)$. Again, this is visualized by the black dot on the $a2$-side of the constraint. The double-headed arrow leaving the black dot specifies that any $e_{ref} \in E^{c2}_{ref}$ should be followed by at least one $a3$ event. The target events in the context of $c2$ are all $a3$ events. Formally: $|\rhd_{e_{ref}}(\partial_{a3}(E))| \geq 1$, i.e., there has to be at least one $a3$ event after e_{ref}.

The two constraints in Figure 3 are just examples. We allow for any constraint that can be specified in terms of the *cardinality of preceding and succeeding target events relative to a collection of reference events*. Therefore, we define the more general notion of *constraint types*.

Definition 4 (Constraint Types). $\mathcal{U}_{CT} = \{X \subseteq \mathbb{N} \times \mathbb{N} \mid X \neq \emptyset\}$ *defines the universe of all possible constraint types. Any element of* $\mathcal{U}_{CT}$ *specifies a non-empty set of pairs of integers: the first integer defines the number of target events before the reference event and the second integer defines the number of target events after the reference event.*

Table 2 shows eight examples of constraint types. Constraint $c1$ is a unary-precedence constraint and constraint $c2$ is a response constraint. The graphical representations of the eight example constraint types are shown in Figure 4. As a shorthand, one arrow may combine two constraints as shown in Figure 5. For example, constraint $c34$ states that after placing an order there is precisely one payment and before a payment there is precisely one order placement.

Table 2. Examples of constraints types (i.e., elements of $\mathcal{U}_{CT}$), inspired by *Declare*. Note that a constraint is defined with respect of a reference event e_{ref}.

constraint	formalization
response	$\{(before, after) \in \mathbb{N} \times \mathbb{N} \mid after \geq 1\}$
unary-response	$\{(before, after) \in \mathbb{N} \times \mathbb{N} \mid after = 1\}$
non-response	$\{(before, after) \in \mathbb{N} \times \mathbb{N} \mid after = 0\}$
precedence	$\{(before, after) \in \mathbb{N} \times \mathbb{N} \mid before \geq 1\}$
unary-precedence	$\{(before, after) \in \mathbb{N} \times \mathbb{N} \mid before = 1\}$
non-precedence	$\{(before, after) \in \mathbb{N} \times \mathbb{N} \mid before = 0\}$
co-existence	$\{(before, after) \in \mathbb{N} \times \mathbb{N} \mid before + after \geq 1\}$
non-co-existence	$\{(before, after) \in \mathbb{N} \times \mathbb{N} \mid before + after = 0\}$

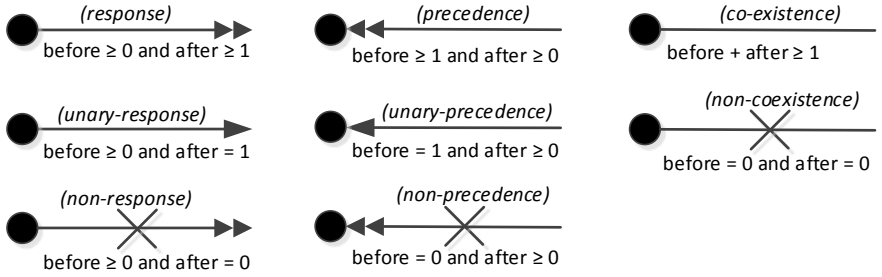

Fig. 4. Graphical notation for the example constraint types defined in Table 2 (example elements of $\mathcal{U}_{CT}$). The dot on the left-hand side of each constraint refers to the *reference events*. *Target events* are on the other side that has no dot. The notation is inspired by *Declare*, but formalized in terms of cardinality constraints rather than LTL.

A *Behavioral Constraint* (BC) model is a collection of activities and constraints (cf. Figure 3).[7]

Definition 5 (Behavioral Constraint Model). *A* behavioral constraint model *is a tuple* $BCM = (A, C, \pi_{ref}, \pi_{tar}, type)$*, where*

- $A \subseteq \mathcal{U}_A$ *is the set of activities (denoted by rectangles),*
- C *is the set of constraints (*$A \cap C = \emptyset$*, denoted by various types of edges),*
- $\pi_{ref} \in C \rightarrow A$ *defines the reference activity of a constraint (denoted by a black dot connecting constraint and activity),*
- $\pi_{tar} \in C \rightarrow A$ *defines the target activity of a constraint (other side of edge), and*
- $type \in C \rightarrow \mathcal{U}_{CT}$ *specifies the type of each constraint (denoted by the type of edge).*

Figure 3 defines the BC model $BCM = (A, C, \pi_{ref}, \pi_{tar}, type)$ with $A = \{a1, a2, a3\}$, $C = \{c1, c2\}$, $\pi_{ref}(c1) = a2$, $\pi_{ref}(c2) = a2$, $\pi_{tar}(c1) = a1$,

[7] Note that BC models depend on the classical instance notion, i.e., the model describes the lifecycle of single instance. OCBC models do not assume a single instance notion. However, we introduce behavioral constraint models to gently introduce the concepts.

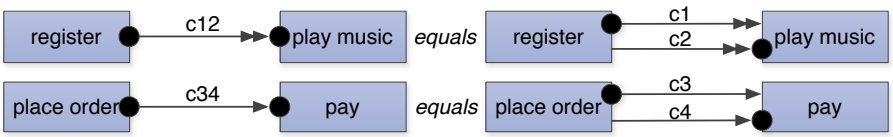

Fig. 5. An arrow with two reference events (•) can be used as a shorthand. Constraint $c12$ ($c34$) corresponds to the conjunction of constraints $c1$ and $c2$ (resp. $c3$ and $c4$).

$\pi_{tar}(c2) = a3$, $type(c1) = \{(before, after) \in \mathbb{N} \times \mathbb{N} \mid before = 1\}$ (unary-precedence), and $type(c2) = \{(before, after) \in \mathbb{N} \times \mathbb{N} \mid after \geq 1\}$ (response). Given a set E of events, we can check whether constraints are satisfied (or not), thus providing a natural link to conformance checking.

Definition 6 (Constraint Satisfaction). *Let $BCM = (A, C, \pi_{ref}, \pi_{tar}, type)$ be a BC model, and $E \subseteq \mathcal{U}_E$ a set of events.*

- *Event set E* satisfies *constraint c if and only if*

$$(|\lhd_{e_{ref}}(\partial_{\pi_{tar}(c)}(E))|, |\rhd_{e_{ref}}(\partial_{\pi_{tar}(c)}(E))|) \in type(c) \text{ for all } e_{ref} \in \partial_{\pi_{ref}(c)}(E)$$

- *Event set E satisfies BCM if and only if E satisfies each constraint $c \in C$.*

The reference activity of a constraint defines the corresponding set of reference events E^c_{ref}. For each reference event it is checked whether the cardinality constraint is satisfied. $\lhd_{e_{ref}}(\partial_{\pi_{tar}(c)}(E))$ are all target events before the reference event e_{ref}. $\rhd_{e_{ref}}(\partial_{\pi_{tar}(c)}(E))$ are all target events after the reference event e_{ref}. Consider, e.g., $c1$ in Figure 3. All $a2$ events are reference events and all $a1$ events are target events. For this example $\lhd_{e_{ref}}(\partial_{\pi_{tar}(c1)}(E))$ is the set of $a1$ events before the selected $a2$ event (e_{ref}). The cardinality of this set should be precisely 1.

In traditional process modeling notations a constraint is defined for one process instance (case) in isolation. This means that the set E in Definition 6 refers to all events corresponding to the same case. As discussed before, the case notion is often too rigid. There may be multiple case-notions at the same time, causing one-to-many or many-to-many relations that cannot be handled using traditional monolithic process models. Moreover, we need to relate events to (data) objects. All these issues are discussed next.

4 Modeling Data Cardinality Constraints

Next to behavior as captured through events, there are also objects that are grouped in classes. Objects may be related and cardinality constraints help to structure dependencies. Entity-Relationship (ER) models [**?**], UML class models [**?**], and Object-Role Models (ORM) [**?**] are examples of notations used for object modeling, often referred to as data modeling. In this paper, we use the simple notation shown in Figure 6(a) to specify *class models*. The notation can be viewed as a subset of such mainstream notations. The only particular feature is that cardinality constraints can be tagged as "always" (□)

or "eventually" ($\Diamond$). For example, for every order order line there is always at most one delivery ($\Box$ 0..1) and eventually (i.e., from some point in time onwards) there should be a corresponding delivery($\Diamond$ 1).

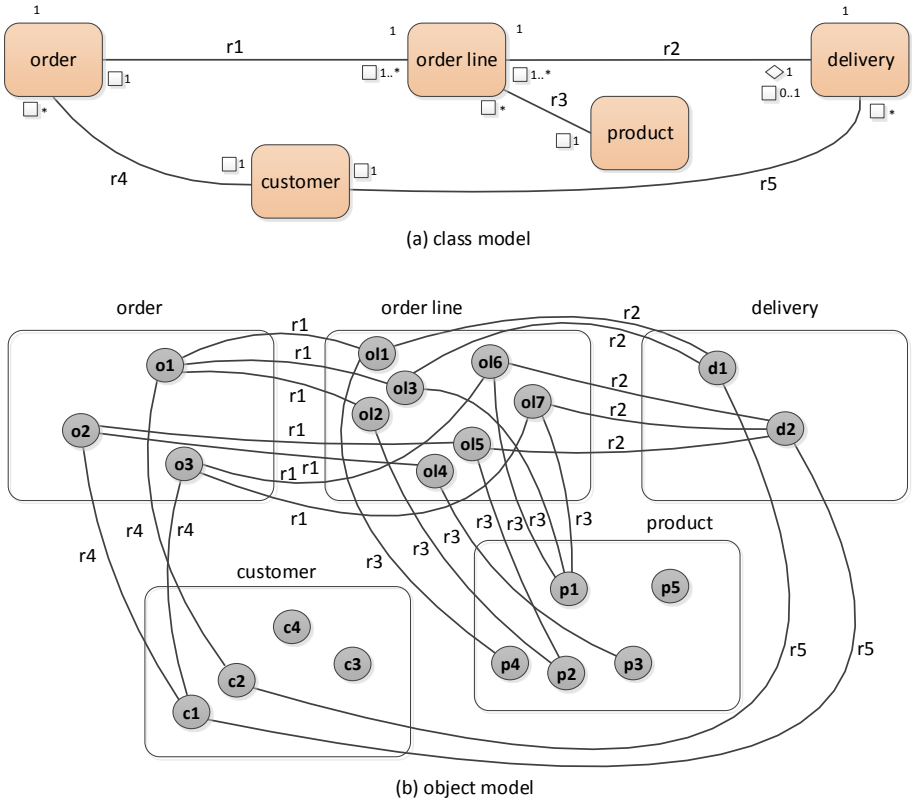

Fig. 6. Example of a class model and corresponding object model.

Definition 7 (Class Model). *A* class model *is a tuple* $ClaM = (OC, RT, \pi_1, \pi_2, \sharp^{\Box}_{src}, \sharp^{\Diamond}_{src}, \sharp^{\Box}_{tar}, \sharp^{\Diamond}_{tar})$*,where*

- OC *is a set of object classes,*
- RT *is a set of relationship types (*$OC \cap RT = \emptyset$*),*
- $\pi_1 \in RT \rightarrow OC$ *gives the source of a relationship,*
- $\pi_2 \in RT \rightarrow OC$ *gives the target of a relationship,*
- $\sharp^{\Box}_{src} \in RT \rightarrow \mathcal{U}_{Card}$ *gives the cardinality of the source of a relationship (the constraint should hold at any point in time as indicated by* $\Box$*),*
- $\sharp^{\Diamond}_{src} \in RT \rightarrow \mathcal{U}_{Card}$ *gives the cardinality of the source of a relationship (the constraint should hold from some point onwards as indicated by* $\Diamond$*),*

- $\sharp^{\Box}_{tar} \in RT \rightarrow \mathcal{U}_{Card}$ *gives the cardinality of the target of a relationship (the constraint should hold at any point in time as indicated by* $\Box$*), and*
- $\sharp^{\Diamond}_{tar} \in RT \rightarrow \mathcal{U}_{Card}$ *gives the cardinality of the target of a relationship (the constraint should hold from some point onwards as indicated by* $\Diamond$*).*

The class model $ClaM = (OC, RT, \pi_1, \pi_2, \sharp^{\Box}_{src}, \sharp^{\Diamond}_{src}, \sharp^{\Box}_{tar}, \sharp^{\Diamond}_{tar})$ depicted in Figure 6(a) has five object classes $OC = \{order, order\ line, delivery, customer, product\}$ and five relationship types $RT = \{r1, r2, r3, r4, r5\}$. Relationship type $r1$ is connecting classes $order$ and $order\ line$: $\pi_1(r1) = order$ and $\pi_2(r1) = order\ line$. For the other relationships types, we have: $\pi_1(r2) = order\ line$, $\pi_2(r2) = delivery$, $\pi_1(r3) = order\ line$, and $\pi_2(r3) = product$, etc.

The notation of Table 1 is extended with $\Box$ ("always") or $\Diamond$ ("eventually") to specify the cardinalities in Figure 6(a). $\sharp^{\Box}_{src}(r1) = \{1\}$, i.e., for each object in class $order\ line$ there is always precisely one corresponding object in $order$. This is indicated by the "$\Box$ 1" annotation on the source side (i.e., the $order$ side of $r1$) in Figure 6(a). $\sharp^{\Box}_{tar}(r1) = \{1, 2, 3, \ldots\}$, i.e., for each object in class $order$ there is always at least one corresponding object in $order\ line$. This is indicated by the "$\Box$ 1..*" annotation on the target side (i.e., the $order\ line$ side) of $r1$. Not shown are $\sharp^{\Diamond}_{src}(r1) = \{1\}$ ("$\Diamond$ 1") and $\sharp^{\Diamond}_{tar}(r1) = \{1, 2, 3, \ldots\}$ ("$\Diamond$ 1..*") as these are implied by the "always" constraints. One the target side of $r2$ in Figure 6(a) there are two cardinality constraints: $\sharp^{\Box}_{tar}(r2) = \{0, 1\}$ and $\sharp^{\Diamond}_{tar}(r2) = \{1\}$. This models that eventually each order line needs to have a corresponding delivery ("$\Diamond$ 1"). However, the corresponding delivery may be created later ("$\Box$ 0..1"). We only show the "eventually" ($\Diamond$) cardinality constraints that are more restrictive than the "always" ($\Box$) cardinalities in the class model. Obviously, $\sharp^{\Diamond}_{src}(r) \subseteq \sharp^{\Box}_{src}(r)$ and $\sharp^{\Diamond}_{tar}(r) \subseteq \sharp^{\Box}_{tar}(r)$ for any $r \in RT$ since constraints that always hold also hold eventually.

Objects can also have attributes and therefore in principle the class model should list the names and types of these attributes. We abstract from object/class attributes in this paper, as well as from the notions of hierarchies and subtyping, but they could be added in a straightforward manner.

A class diagram defines a "space" of possible *object models*, i.e., concrete collections of objects and relations instantiating the class model.

Definition 8 (Object Model). *$\mathcal{U}_O$ is the universe of object identifiers. An object model for class model $ClaM = (OC, RT, \pi_1, \pi_2, \sharp^{\Box}_{src}, \sharp^{\Diamond}_{src}, \sharp^{\Box}_{tar}, \sharp^{\Diamond}_{tar})$ is a tuple $OM = (Obj, Rel, class)$, where:*

- *$Obj \subseteq \mathcal{U}_O$ is the set of objects,*
- *$Rel \subseteq RT \times Obj \times Obj$ is the set of relations,*
- *$class \in Obj \rightarrow OC$ maps objects onto classes.*

$\mathcal{U}_{OM}$ is the universe of object models.

Figure 6(b) shows an object model $OM = (Obj, Rel, class)$. The objects are depicted as grey dots: $Obj = \{o1, o2, o3, ol1, ol2, \ldots, ol7, d1, d2, c1, \ldots, c4, p1, \ldots, p5\}$. There are three objects belonging to object class $oc1$, i.e., $class(o1) = class(o2) = class(o3) = order$. There are seven relations corresponding to relationship $r1$, e.g., $(r1, o1, ol1) \in Rel$ and $(r1, o2, ol5) \in Rel$.

Note that objects and events are represented by unique identifiers. This allows us to refer to a specific object or event. Even two events or objects with the same properties are still distinguishable by their identity.

The cardinalities specified in the class model should be respected by the object model. For example, for each object in class $order\ line$ there is precisely one corresponding object in $order$ according to $r1$. A *valid* object model complies with the "always" ($\Box$) cardinalities in the class model. A valid model is also *fulfilled* is also the possibly stronger "eventually" ($\Diamond$) cardinality constraints are satisfied.

Definition 9 (Valid Object Model). *Let* $ClaM = (OC, RT, \pi_1, \pi_2, \sharp^{\Box}_{src}, \sharp^{\Diamond}_{src}, \sharp^{\Box}_{tar}, \sharp^{\Diamond}_{tar})$ *be a class model and* $OM = (Obj, Rel, class) \in \mathcal{U}_{OM}$ *be an object model.* OM *is valid for* $ClaM$ *if and only if*

- *for any* $(r, o_1, o_2) \in Rel$*:* $class(o_1) = \pi_1(r)$ *and* $class(o_2) = \pi_2(r)$*,*
- *for any* $r \in RT$ *and* $o_2 \in \partial_{\pi_2(r)}(Obj)$*, we have that* [8]

$$|\{o_1 \in Obj \mid (r, o_1, o_2) \in Rel\}| \in \sharp^{\Box}_{src}(r), \text{ and}$$

- *for any* $r \in RT$ *and* $o_1 \in \partial_{\pi_1(r)}(Obj)$*, we have that*

$$|\{o_2 \in Obj \mid (r, o_1, o_2) \in Rel\}| \in \sharp^{\Box}_{tar}(r)$$

A valid objected model is also fulfilled if the stronger cardinality constraints hold (these are supposed to hold eventually):

- *for any* $r \in RT$ *and* $o_2 \in \partial_{\pi_2(r)}(Obj)$*, we have that*

$$|\{o_1 \in Obj \mid (r, o_1, o_2) \in Rel\}| \in \sharp^{\Diamond}_{src}(r), \text{ and}$$

- *for any* $r \in RT$ *and* $o_1 \in \partial_{\pi_1(r)}(Obj)$*, we have that*

$$|\{o_2 \in Obj \mid (r, o_1, o_2) \in Rel\}| \in \sharp^{\Diamond}_{tar}(r)$$

The object model in Figure 6(b) is indeed valid. If we would remove relation $(r1, o1, ol1)$, the model would no longer be valid (because an order line should always have a corresponding order). Adding a relation $(r1, o2, ol1)$ would also destroy validity. Both changes would violate the "$\Box$ 1" constraint on the source side of $r1$. The object model in Figure 6(b) is not fulfilled because the "$\Diamond$ 1" constraint on the target side of $r2$ does not hold. Order lines $ol2$ and $ol4$ do not (yet) have a corresponding delivery. Adding deliveries for these order lines and adding the corresponding relations would make the model fulfilled.

Definition 9 only formalizes simple cardinality constraints involving a binary relation and abstracting from attribute values. In principle more sophisticated constraints could be considered: the object model OM is simply checked against a class model $ClaM$. For example, the Object Constraint Language (OCL) [**?**] could be used to define more refined constraints.

[8] $\partial_{oc}(Obj) = \{o \in Obj \mid class(o) = oc\}$ denotes the whole set of objects in class oc.

5 Object-Centric Behavioral Constraints

In Section 3, we focused on control-flow modeling and formalized behavioral constraints *without* considering the structure of objects. In Section 4, we focused on structuring objects and formalized cardinality constraints on object models (i.e., classical data modeling). In this section, we combine *both perspectives* to fully address the challenges described in the introduction.

5.1 Object-Centric Event Logs

First, we formalize the notion of an event log building on the event notion introduced in Definition 1. An event log is a collection of events that belong together, i.e., they belong to some "process" where many types of objects/instances may interact. Next to scoping the log, we also relate events to objects. Note that the same event may refer to multiple objects and one object may be referred to by multiple events.

Definition 10 (Event Log). *An* event log *is a tuple* $L = (E, act, attr, EO, om, \preceq)$*, where*

- $E \subseteq \mathcal{U}_E$ *is a set of events,*
- $act \in E \rightarrow \mathcal{U}_A$ *maps events onto activities,*
- $attr \in E \rightarrow (\mathcal{U}_{Attr} \not\rightarrow \mathcal{U}_{Val})$ *maps events onto a partial function assigning values to some attributes,*
- $EO \subseteq E \times \mathcal{U}_O$ *relates events to sets of object references,*
- $om \in E \rightarrow \mathcal{U}_{OM}$ *maps each event to the object model directly after the event took place, and*
- $\preceq \subseteq E \times E$ *defines a total order on events.*

In the context of an event L, each event e is associated with object model $OM_e = (Obj_e, Rel_e, class_e) = om(e)$. In the remainder, we refer directly to Obj_e, Rel_e, $class_e$ for $e \in E$ if the context is clear.

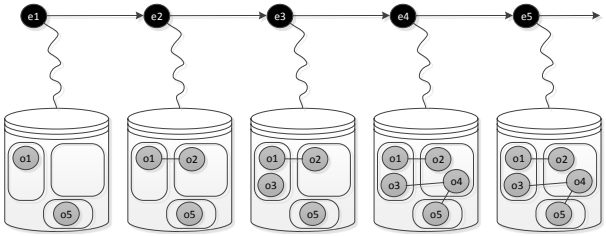

Fig. 7. Each event e refers to the object model right after e occurred: $OM_e = (Obj_e, Rel_e, class_e) = om(e)$.

Figure 7 illustrates the evolution of the object model. After the occurrence of some event e objects may have been added (we assume monotonicity), and relationships may

have been added or removed. Event e may refer to objects through relation EO and these objects need to exist, i.e., for all $(e, o) \in EO$: $o \in Obj_e$. We assume that objects cannot be removed at a later stage to avoid referencing non-existent objects. Objects can be marked as deleted but cannot be removed (e.g, by using an attribute or relation).

The event log provides a *snapshot of the object model after each event*. This triggers the question: Can the object model be changed in-between two subsequent events? If no such changes are possible, then the object model before an event is the same as the object model after the previous event. If we would like to allow for updates in-between events, then these could be recorded in the log. Events referring to some artificial activity *update* could be added to signal the updated object model. We could also explicitly add a snapshot of the object model just before each event. In the remainder, we only consider the snapshot OM_e after each event $e \in E$.

Note that Definition 10 calls for event logs different from the standard *XES format*. XES (`www.xes-standard.org`), which is supported by the majority of process mining tools, assumes a case notion (i.e., each event refers to a process instance) and does not keep track of object models. Therefore, we defined the *XOC format* which is basically an XML version of Definition 10 (see Section 7). In Section 8 we show that such event logs can indeed be extracted from today's information systems.

5.2 OCBC Models

Next, we define *Object-Centric Behavioral Constraint* (OCBC) models. Through a combination of control-flow modeling and data/object modeling, we relate behavior and structure. The BC models from Section 3 are connected to the class models of Section 4 to provide the integration needed.

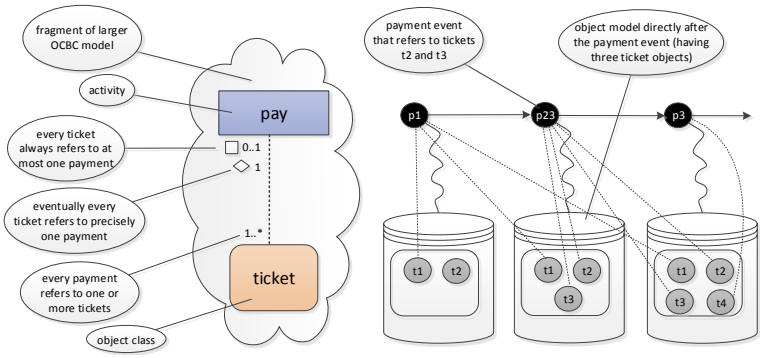

Fig. 8. Illustrating cardinality constraints $\sharp_A^{\square}$, $\sharp_A^{\diamond}$, and $\sharp_{OC}$.

A key ingredient is that events and objects are related as illustrated in Figure 8. Payment activity $p1$ refers to ticket $t1$, activity $p23$ refers to tickets $t2$ and $t3$, and activity $p3$ refers to ticket $t4$. Figure 8 shows three example constraints: "$\square$ 0..1" (every

ticket always refers to at most one payment), "$\Diamond$ 1" (eventually every ticket refers to precisely one payment), and "1..*" (every payment refers to one or more tickets).

Definition 11 (Object-Centric Behavioral Constraint Model). *An* object-centric behavioral constraint model *is a tuple* $OCBCM = (BCM, ClaM, AOC, \sharp^{\Box}_{A}, \sharp^{\Diamond}_{A}, \sharp_{OC}, crel)$*, where*

- $BCM = (A, C, \pi_{ref}, \pi_{tar}, type)$ *is a BC model (Definition 5),*
- $ClaM = (OC, RT, \pi_1, \pi_2, \sharp^{\Box}_{src}, \sharp^{\Diamond}_{src}, \sharp^{\Box}_{tar}, \sharp^{\Diamond}_{tar})$ *is a class model (Definition 7),*
- A*,* C*,* OC *and* RT *are pairwise disjoint (no name clashes),*
- $AOC \subseteq A \times OC$ *is a set of relations between activities and object classes,*
- $\sharp^{\Box}_{A} \in AOC \rightarrow \mathcal{U}_{Card}$ *gives the cardinality of the source of a relation linking an activity and an object class (activity side, the constraint should hold at any point in time as indicated by* $\Box$*),*
- $\sharp^{\Diamond}_{A} \in AOC \rightarrow \mathcal{U}_{Card}$ *gives the cardinality of the source of a relation linking an activity and an object class (activity side, the constraint should hold from some point onwards as indicated by* $\Diamond$*),*
- $\sharp_{OC} \in AOC \rightarrow \mathcal{U}_{Card}$ *gives the cardinality of the target of a relation linking an activity and an object class (object-class side), and*
- $crel \in C \rightarrow OC \cup RT$ *is the constraint relation satisfying the following conditions for each* $c \in C$*:*
  - $\{(\pi_{ref}(c), oc), (\pi_{tar}(c), oc)\} \subseteq AOC$ *if* $crel(c) = oc \in OC$*, and*
  - $\{(\pi_{ref}(c), \pi_1(r)), (\pi_{tar}(c), \pi_2(r))\} \subseteq AOC$ *or* $\{(\pi_{ref}(c), \pi_2(r)), (\pi_{tar}(c), \pi_1(r))\} \subseteq AOC$ *if* $crel(c) = r \in RT$*.*

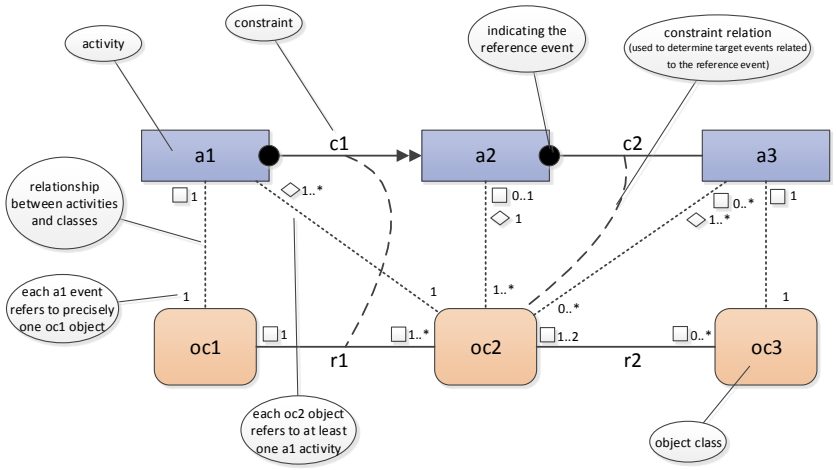

Fig. 9. An example model illustrating the main ingredients of an OCBC model.

An *Object-Centric Behavioral Constraint model* (OCBC model) includes a behavioral constraint model (to model behavior) and a class model (to model objects/data). These are related through relation AOC and functions $\sharp_A^{\Box}$, $\sharp_A^{\Diamond}$, $\sharp_{OC}$, and $crel$. We use Figure 9 to clarify these concepts.

AOC relates activities and object classes. In Figure 9, $AOC = \{(a1, oc1), (a1, oc2), (a2, oc2), (a3, oc2), (a3, oc3)\}$. For example, $a1$ may potentially refer to $oc1$ and $oc2$ objects, but not to $oc3$ objects because $(a1, oc3) \notin AOC$. Recall that in an event log L there is a many-to-many relationship between events and objects ($EO \subseteq E \times \mathcal{U}_O$) constrained by AOC.

Functions $\sharp_A^{\Box}$, $\sharp_A^{\Diamond}$, and $\sharp_{OC}$ define possible cardinalities, similar to cardinality constraints in a class model. Functions $\sharp_A^{\Box}$ and $\sharp_A^{\Diamond}$ define how many events there need to be for each object. Since the object model is evolving, there are two types of constraints: constraints that should hold *at any point in time* from the moment the object exists ($\sharp_A^{\Box}$) and constraints that should *eventually* hold $\sharp_A^{\Diamond}$. Function $\sharp_{OC}$ defines how many objects there need to be for each event when the event occurs (specified by EO).

As indicated by the "$\Box$ 1" annotation on the $a1$-side of the line connecting activity $a1$ and object class $oc1$, there is precisely one $a1$ event for each $oc1$ object (from the moment it exists): $\sharp_A^{\Box}(a1, oc1) = \{1\}$. As indicated by the "$\Diamond$ 1..*" on the $a1$-side of the line connecting activity $a1$ and object class $oc2$, we have that $\sharp_A^{\Diamond}(a1, oc2) = \{1, 2, \ldots\}$. This means that *eventually* each $oc2$ object refers to at least one $a1$ activity. Note that an $oc2$ object does not need to have a corresponding $a1$ event when it is created. However, adding a new $oc2$ object implies the occurrence of at least one corresponding $a1$ event to satisfy the cardinality constraint "$\Diamond$ 1..*", i.e., an *obligation* is created. If the annotation "$\Box$ 1..*" would have been used (instead of "$\Diamond$ 1..*"), then the creation of any $oc2$ object needs to coincide with a corresponding $a1$ event, because the cardinality constraints should always hold ($\Box$) and not just eventually ($\Diamond$).

As indicated by the "1" annotation on the $oc2$-side of the line connecting activity $a1$ and object class $oc2$, we then have that $\sharp_{OC}(a1, oc2) = \{1\}$. This means that each $a1$ activity refers to precisely one $oc2$ object.

Let's now consider relation $(a2, oc2) \in AOC$. There should be at most one $a2$ event for each $oc2$ object from the moment it exists: $\sharp_A^{\Box}(a2, oc2) = \{0, 1\}$. Eventually there should be precisely one $a2$ event for each $oc2$ object: $\sharp_A^{\Diamond}(a2, oc2) = \{1\}$. $\sharp_{OC}(a2, oc2) = \{1, 2, \ldots\}$ indicates that each $a2$ event refers to at least one $oc2$ object.

Annotations of the type "$\Diamond$ 0..*" and "$\Box$ 0..*" are omitted from the diagram because these impose no constraints. Also implied constraints can be left out, e.g., "$\Box$ 1..*" implies "$\Diamond$ 1..*".

Function $crel$ defines the *scope* of each constraint thereby relating reference events to selected target events. $crel(c)$ specifies how events need to be correlated when evaluating constraint c. This is needed because we do not assume a fixed case notion and different entities may interact. As illustrated by Figure 10 we basically consider two types of constraints. In both cases we navigate through the object model to find target events for a given reference event. Figures 11 and 12 illustrate how to locate target events. For each reference event we need the set of all target events in order to check the cardinality constraint.

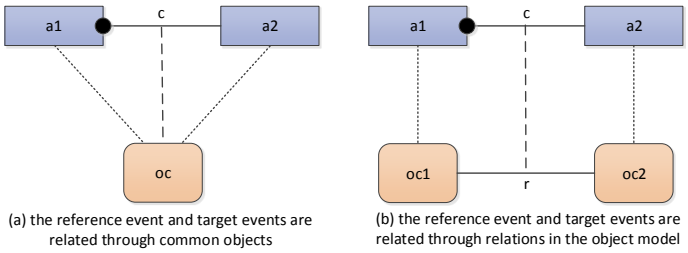

Fig. 10. Two types of constraint relations: (a) $crel(c) = oc \in OC$, i.e., the target events are related to the reference event through shared objects of the class oc, (b) $crel(c) = r \in RT$, i.e., the target events are related to the reference event through relations of type r (in any direction).

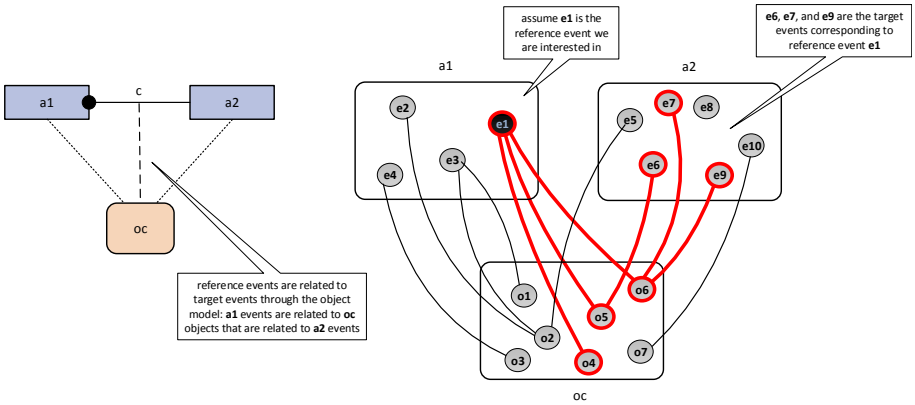

Fig. 11. Given a reference event for a constraint with $crel(c) = oc \in OC$ we navigate to the target events through shared object references.

If $crel(c) = oc \in OC$, then the behavioral constraint is based on object class oc. In Figure 9, $crel(c2) = oc2$. This means that the target events for constraint $c2$ need to be related to the reference events through objects of class $oc2$. Let e_{ref} be the reference event for constraint $c2$. e_{ref} refers to 1 or more $oc2$ objects. The target events of e_{ref} for $c2$ are those $a3$ events referring to one of these objects.

If $crel(c) = r \in RT$, then the target events are related to the reference event through relations of type r in the object model. Relation r can be traversed in both directions. In Figure 9, $crel(c1) = r1$ indicating that reference events are related to target events through relationship $r1$. Let e_{ref} be the reference $a1$ event for constraint $c1$. e_{ref} refers to $oc1$ objects that are related to $oc2$ objects through $r1$ relations. The target events of e_{ref} for $c1$ are those $a2$ events referring to one of these $oc2$ objects.

We have now introduced all the modeling elements used in Figure 2. Note that *create order* activities are related to *pick item* activities through the relationship connecting class *order* with class *order line*.

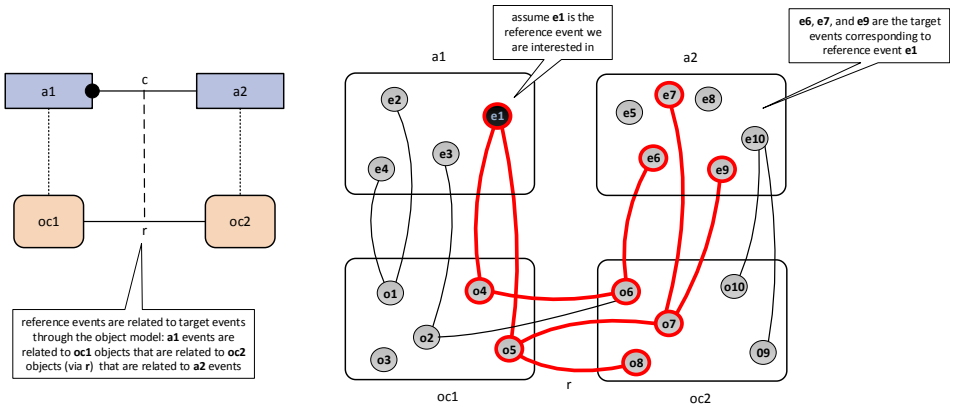

Fig. 12. Given a reference event for a constraint with $crel(c) = r \in RT$ we navigate to the target events through relation r in the object model.

5.3 Discussion

The graphical notation introduced (e.g., like in Figure 2) fully defines an OCBC model. To illustrate this let us consider a completely different example.

Figure 13 models a hiring process. An organization may create a position. People can apply for such a position, but need to register first. Applications for a position are only considered in the period between opening the position and closing the application process for the position. An application may be followed by at most five reference checks and at most two interviews. In the end one person is selected and subsequently hired for the position.

There are four object classes in the OCBC model: *person*, *application*, *position*, and *employee*. The cardinality constraints in Figure 13 show that: each application always refers to precisely one person and one position, each person eventually applies for some position, for every position there will eventually be an application, each employee refers to precisely one application and position, each application refers to at most one employee, and each position will eventually refer to one employee.

There is a one-to-one correspondence between registrations (activity *register*) and persons (class *person*). Activities *apply*, check *reference*, and *interview* each refer to the class *application*. Activity *apply* creates one new *application* object. Activities *open pos.*, check *close pos.*, and *select* each refer to the class *position*. Activity *open pos.* creates one new *position* object. There is also a one-to-one correspondence between hirings (activity *hire*) and employees (class *employee*).

Let us now consider the constraints in more detail:

- Constraint $c1$ specifies that every reference check should be preceded by precisely one corresponding application (unary-precedence constraint).
- Constraint $c2$ specifies that every interview should be preceded by precisely one corresponding application (unary-precedence).

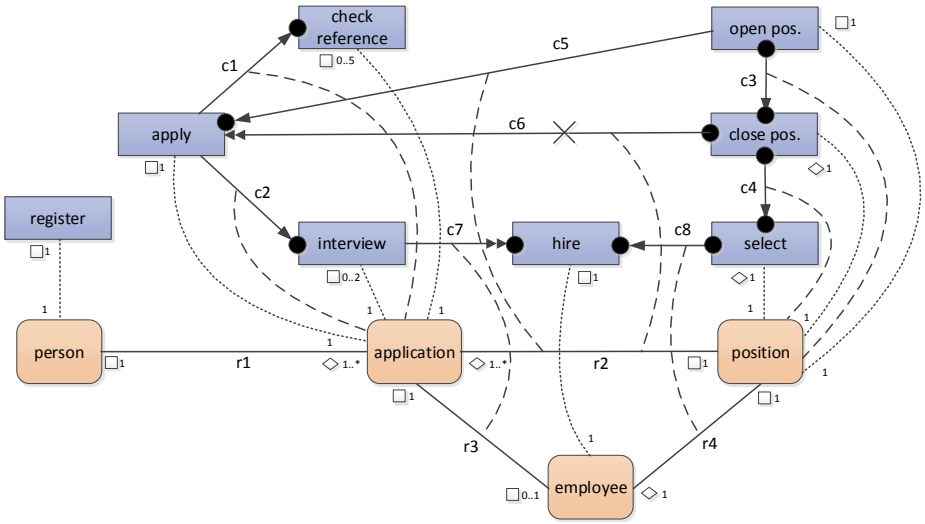

Fig. 13. An OCBC model modeling a hiring process.

- Constraint $c3$ combines a unary-response and a unary-precedence constraint stating that the opening a a position should be followed by the closing of the application process and the closing should be preceded by the opening of the position.
- Constraint $c4$ also combines a unary-response and a unary-precedence constraint stating that the two activities are executed in sequence.
- Constraint $c5$ specifies that applications for a position need to be preceded by the opening of that position.
- Constraint $c6$ specifies that after closing a position there should not be any new applications for this position (non-response constraint).
- Constraint $c7$ specifies that every hire needs to be preceded by at least one interview with the candidate applying for the position (precedence constraint).
- Constraint $c8$ again combines a unary-response and a unary-precedence constraint stating that the two activities are executed in sequence.

It is important to note that the constraints are based on the object model and that there is not a single instance notion. To illustrate this consider the BPMN model in Figure 14 which models the lifecycles of persons, positions, applications, and employees in separate diagrams. The BPMN model looks very simple, but fails to capture dependencies between the different entities. Consider for example constraints $c5$, $c6$, $c7$, and $c8$ in the OCBC model of Figure 13. The BPMN model does *not* indicate that there is a one to many relationship between positions and applications, and does *not* show that one can only apply if the corresponding position is opened but not yet closed. The BPMN model does *not* indicate that only one person is hired per position and that the person to be hired should have registered, applied, and had at least one interview. The BPMN model does *not* indicate that employees are hired after the completion of the

selection process. Note that the same person could apply for multiple positions and many people may apply for the same position. Obviously this cannot be captured using a single process instance (case) notion.

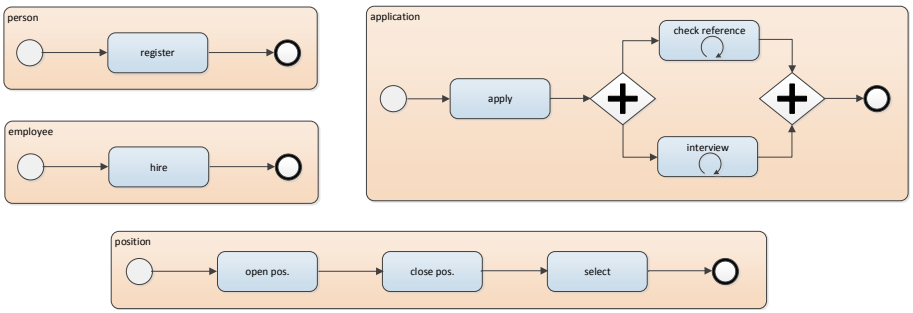

Fig. 14. An attempt to capture the OCBC model of Figure 13 in terms of four BPMN models. The relations with the overall data model and interactions between the different entities are no longer visible. For example, insights like "one can only apply if the corresponding position is opened but not yet closed" and "only people that had an interview can be hired" get lost.

Comparing Figure 13 and Figure 14 reveals that modeling the lifecycles of entities separately, like in artifact-centric approaches, is not sufficient to capture the real process. The individual lifecycles are simple, but fail to reveal the interplay between persons, positions, applications, and employees.

It is essential to understand that the scoping of events considered in a constraint is done through the object model. This provides a tight integration between behavior and structure. Moreover, the approach is much more general and more expressive than classical approaches where events are correlated through cases. Normally, process models (both procedural and declarative) describe the lifecycle of a process instance (i.e., case) in isolation. This implies that events are partitioned based on case identifiers and different cases cannot share events. Hence, one-to-many and many-to-many relationships cannot be modeled (without putting instances in separate subprocesses, artifacts or proclets). In fact, more complicated forms of interaction cannot be handled.

Note that traditional single-instance modeling approaches can still be mimicked by using an object model having one object class $case$ and $crel(c) = case$ for each constraint c. Figure 15 sketches this situation and illustrates that the classical view on process behavior is every limiting, since complex relationships cannot be captured, and the link to data/object models is missing.

6 Conformance Checking Using OCBC Models

Given an event log, an object model, and an object-centric behavioral constraint model, we want to check whether reality (in the form of an event log L and an object model

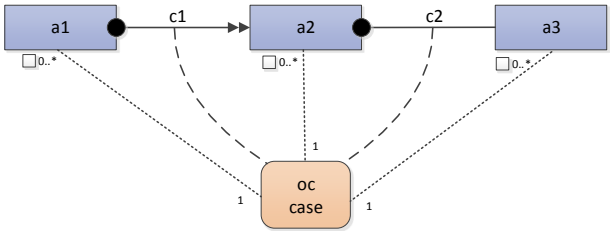

Fig. 15. An OCBC model mimicking the classical situation where behavior needs to be straightjacketed in isolated process instances (i.e., cases).

OM) conforms to the model $OCBCM$. We identify nine types of possible conformance problems. Most of these problems are not captured by existing conformance checking approaches [**?**,**?**,**?**,**?**].

First, we implicitly provide operational semantics for OCBC models by defining a *conformance relation* between event log and model.

Definition 12 (Conformance). *Let* $OCBCM = (BCM, ClaM, AOC, \sharp_A^{\Box}, \sharp_A^{\Diamond}, \sharp_{OC}, crel)$ *be an OCBC model, with* $BCM = (A, C, \pi_{ref}, \pi_{tar}, type)$ *and* $ClaM = (OC, RT, \pi_1, \pi_2, \sharp_{src}^{\Box}, \sharp_{src}^{\Diamond}, \sharp_{tar}^{\Box}, \sharp_{tar}^{\Diamond})$. *Let* $L = (E, act, attr, EO, om, \preceq)$ *be an event log.*

Event log L conforms *to the object-centric behavioral constraint model* $OCBCM$ *if and only if:*

- ***There are no Type I problems (validity of object models):*** *for any* $e \in E$*: object model* $OM_e = (Obj_e, Rel_e, class_e)$ *is valid for* $ClaM$ *(this includes checking the* $\Box$*-cardinality constraints that should always hold as stated in Definition 9),*
- ***There are no Type II problems (fulfilment):*** *there is an event* $e_f \in \trianglerighteq_e(E)$ *such that for any* $e' \in \trianglerighteq_{e_f}(E)$*:* $OM_e = (Obj_e, Rel_e, class_e)$ *is also fulfilled (this involves checking the* $\Diamond$*-cardinality constraints that should eventually hold as stated in Definition 9),*
- ***There are no Type III problems (monotonicity):*** *for any* $e_1, e_2 \in E$ *such that* $e_1 \prec e_2$*:* $Obj_{e_1} \subseteq Obj_{e_2}$ *and* $class_{e_1} \subseteq class_{e_2}$ *(objects do not disappear or change class in-between events).*
- ***There are no Type IV problems (activity existence):*** $\{act(e) \mid e \in E\} \subseteq A$ *(all activities referred to by events exist in the behavioral model),*
- ***There are no Type V problems (object existence):*** *for all* $(e, o) \in EO$*:* $o \in Obj_e$ *(all objects referred to by an event exist in the object model when the event occurs),*[9]
- ***There are no Type VI problems (proper classes):*** $\{(act(e), class_e(o)) \mid (e, o) \in EO\} \subseteq AOC$ *(events do not refer to objects of unrelated classes).*
- ***There are no Type VII problems (right number of events per object):*** *for any* $(a, oc) \in AOC$*,* $e \in E$*, and* $o \in \partial_{oc}(Obj_e)$*:*

[9] Combined with the earlier requirement, this implies that these objects also exist in later object models.

  - *for any* $e' \in \trianglerighteq_e(E)$*:* $|\{e'' \in \partial_a(\trianglelefteq_{e'}(E)) \mid (e'', o) \in EO\}| \in \sharp_A^{\Box}(a, oc)$ *(each object* o *of class* oc *has the required number of corresponding* a *events),*
  - *there exists a future event* $e_f \in \trianglerighteq_e(E)$ *such that for any* $e' \in \trianglerighteq_{e_f}(E)$*:* $|\{e'' \in \partial_a(\trianglelefteq_{e'}(E)) \mid (e'', o) \in EO\}| \in \sharp_A^{\Diamond}(a, oc)$ *(each object* o *of class* oc *eventually has the required number of corresponding* a *events),*
- ***There are no Type VIII problems (right number of objects per event):*** *for any* $(a, oc) \in AOC$*,* $e \in \partial_a(E)$*:* $|\{o \in \partial_{oc}(Obj_e) \mid (e, o) \in EO\}| \in \sharp_{OC}(a, oc)$ *(each event* e *corresponding to activity* a *has the required number of corresponding objects of class* oc*).*
- ***There are no Type IX problems (behavioral constraints are respected):*** *for each constraint* $c \in C$ *and reference event* $e_{ref} \in \partial_{\pi_{ref}(c)}(E)$*: there exists a future event* $e_f \in E$ *such that for any* $e' \in \trianglerighteq_{e_f}(E)$*:* $(|\triangleleft_{e_{ref}}(E_{tar})|, |\triangleright_{e_{ref}}(E_{tar})|) \in type(c)$ *where*
  - $E_{tar} = \{e_{tar} \in \partial_{\pi_{tar}(c)}(E) \mid \exists_{o \in \partial_{oc}(Obj_{e'})} \{(e_{ref}, o), (e_{tar}, o)\} \subseteq EO\}$ *if* $crel(c) = oc \in OC$*,*
  - $E_{tar} = \{e_{tar} \in \partial_{\pi_{tar}(c)}(E) \mid \exists_{o_1, o_2 \in Obj_{e'}} (\{(r, o_1, o_2), (r, o_2, o_1)\} \cap Rel_{e'} \neq \emptyset) \wedge \{(e_{ref}, o_1), (e_{tar}, o_2)\} \subseteq EO\}$ *if* $crel(c) = r \in RT$*.*

Any event log L that exhibits none of the nine problems mentioned is conforming to $OCBCM$. Therefore, one can argue that Definition 12 provides operational semantics to OCBC models. However, the ultimate goal is not to provide semantics, but to check conformance and provide useful diagnostics. By checking conformance using Definition 12, the following four broad classes of problems may be uncovered:

- Type I, II, and III problems are related to the object models attached to the events (e.g., object models violating cardinality constraints).
- Type IV, V, and VI problems are caused by events referring to things that do not exist (e.g., non-existing activities or objects).
- Type VII and VIII problems refer to violations of cardinality constraints between activities and object classes.
- Type IX problems refer to violations of the behavioral constraints (e.g., a violation of a response or precedence constraint).

The first two categories (Type I-VI problems) types are more of a bookkeeping nature and relatively easy to understand. The two categories (VII, VIII, IX problems) are related to the more subtle interplay between activities, objects, relations, and the behavior over time. These are more interesting, but also quite difficult to understand. Therefore, we elaborate on Type VII, VIII, IX problems.

Figure 16 shows a situation with problems of Type VII and Type VIII. Object $t3$ has twee corresponding payment events ($p1$ and $p2$), thus violating the "$\Box$ 0..1" annotation. Object $t5$ has no corresponding payment events, thus violating the "$\Diamond$ 1" annotation (assuming there is no corresponding payment in the future). Event $p3$ has no corresponding payment events, thus violating the "1..*" annotation. Note that the object model is evolving while the process is executed. This is not shown in Figure 16, i.e., the diagram should be viewed as a snapshot of the process after four payment events.

Figure 17 shows a situation with problems of Type IX. All $a2$ events should have precisely one preceding $a1$ event that is related through relation r. Note that in principle the object model is evolving, but let us assume that all seven events have the object

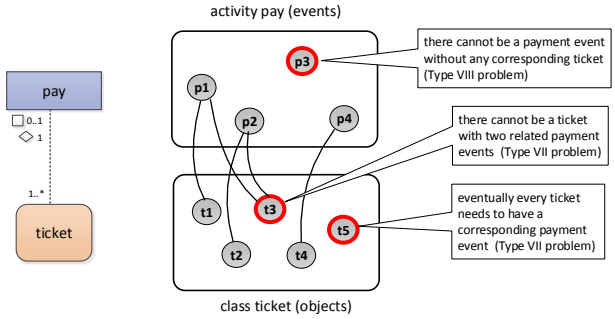

Fig. 16. An illustration of Type VII and VIII problems (all related to violations of cardinality constraints between activities and object classes).

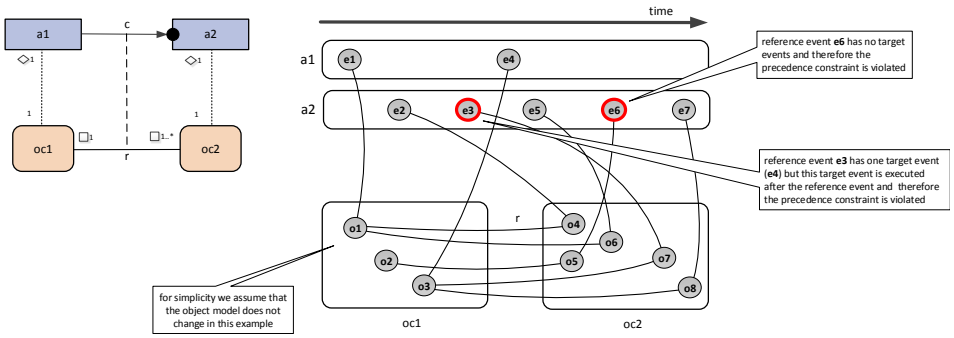

Fig. 17. An illustration of Type IX problems. The $a1$ and $a2$ events are executed in the order indicated (from left to right). The object model is assumed to remain invariant during the execution of the events (to simplify the explanation). Constraint c is violated for two of the five reference events: both $e3$ and $e6$ have no corresponding $a1$ event that occurred earlier.

model shown at the lower part of Figure 17. As stated in Definition 12, there should be an event e_f after which the constraint holds for any event e' and corresponding object model $OM_{e'}$. Note that the two cases in the last condition of Definition 12 correspond to the two constraint relations depicted in Figure 10. For each reference event e_{ref}, the corresponding set of target events E_{tar} is determined by following the links through the object model. For each e_{ref}, the cardinalities are checked: $(|\lhd_{e_{ref}}(E_{tar})|, |\rhd_{e_{ref}}(E_{tar})|) \in type(c)$. Hence, it is possible to identify the reference events for which the constraint is violated.

For the situation depicted in Figure 17: $type(c) = \{(before, after) \in \mathbb{N} \times \mathbb{N} \mid before = 1\}$, i.e., there should be precisely one target ($a1$) event preceding each reference ($a2$) event related through r. Consider $e2 = e_{ref}$ as reference event: $E_{tar} = \{e1\}$ and target event $e1$ occurs indeed before $e2$. Hence, no problem is discovered for $e2$. Next we consider $e3 = e_{ref}$ as reference event: $E_{tar} = \{e4\}$, but target event $e4$ occurs

after $e3$. Hence, $e3$ has no preceding target event signaling a violation of constraint c for reference event $e3$. If we consider $e5 = e_{ref}$ as reference event, we find no problem because $E_{tar} = \{e1\}$ and target event $e1$ occurs indeed before $e5$. If we consider $e6 = e_{ref}$ as reference event, we find again a problem because $E_{tar} = \emptyset$, so no target event occurs before $e6$. If we consider $e7 = e_{ref}$ as reference event, we find no problem because $E_{tar} = \{e4\}$ and target event $e4$ occurs before after $e7$. Hence, we find two reference event ($e3$ and $e6$) for which constraint c in Figure 17 does not hold.

Definition 12 not only provides operational semantics for the graphical notation introduced in this paper, but also characterizes a wide range of conformance problems. Following the classification of problems used in Definition 12, we mention some of the diagnostics possible (all supported in our implementation).

⟨TODO: Needs to be refined by Guangming Li. There is also a mismatch with the tool. There are lists with 7, 9 and 5 problem types. See also naming problems in the tool.⟩

1. **Diagnostics for Type I problems (validity of object models):** Display the invalid object models with the deviating elements. One can know the reason leading to the invalidity, e.g., $\Box$-cardinality constraints that make the object models invalid.
2. **Diagnostics for Type II problems (fulfilment):** Report $\Diamond$-cardinality constraints that do not hold at the end of the log as well as the deviating elements in the object model.
3. **Diagnostics for Type III problems (monotonicity):** Report objects that disappear or change class over time.
4. **Diagnostics for Type IV problems (activity existence):** Display the activities appearing in the log and not in the model as well as the corresponding events in the event log.
5. **Diagnostics for Type V problems (object existence):** Present the objects which are referred to by events and do not exist in the corresponding object models.
6. **Diagnostics for Type VI problems (proper classes):** Report the events that refer to classes they should not refer to.
7. **Diagnostics for Type VII problems (right number of events per object):** Return each violated AOC relation, i.e., each (a, oc) connection if objects in oc do not (always/eventually) have the required number of a events. One can see the deviating objects and their corresponding events for each AOC relation.
8. **Diagnostics for Type VIII problems (right number of objects per event):** Return each violated AOC relation, i.e., each (a, oc) connection if a events do not refer to the specified number of objects in class oc. One can see the deviating events and their corresponding objects for each AOC relation.
9. **Diagnostics for Type IX problems (behavioral constraints are respected):** Report the constraints that are violated. List the violated constraints as well as corresponding reference events. Per violated constraint one can view the reference events and their corresponding target events.

The types diagnostics and checks needed are very different from existing conformance checking approaches. Most of the conformance checking approaches [**?**,**?**,**?**,**?**,**?**] only consider control-flow and are unable to uncover the above problems. Recently, conformance checking approaches based on alignments have been extended to also check conformance with respect to the data perspective [**?**,**?**]. However, these do not consider a data model and focus on one instance at a time.

Constraints may be *temporarily violated* while the process is running [**?**,**?**]. Consider for example a response constraint involving activities a and b: after executing activity a the constraint is temporarily violated until activity b is executed. This notion exists in any modeling language where process instances need to terminate and is not limited to declarative languages. Interestingly, the addition of an object may also create temporarily violated and permanently violated constraints. Consider Figure 2 again. Adding an *order line* object without creating a corresponding *order* results in a permanent violation. However, adding an *order line* while also creating an *order* creates a cascade of obligations: the obligation to have a *delivery* object, the obligation to have a *pick item* event, and the obligation to have a *wrap item* event. The corresponding three "$\Diamond$ 1" cardinalities are temporarily violated, but can still be satisfied in the future. Implicitly, there is also the obligation to have a corresponding *deliver items* event in the future.

⟨TODO: Marco: Drop the paper below? Is it still valid/needed?⟩

Interestingly, conformance over OCBC models can be checked very efficiently. In particular, each of the requirements in Definition 12 can be formalized as a boolean, SQL-like query over the input log. The final result is obtained by conjoining all the obtained answers. This means that the data complexity of conformance checking[10] is in AC_0. Recall that AC_0 is strictly contained in LOGSPACE, and corresponds to the complexity of SQL query answering over a relational database, measured in the size of the database only.

7 Implementation

We have implemented the conformance checking approach just described. However, because we uses models and event logs very different from existing approaches, we also had to implement an OCBC model editor and develop an infrastructure to manage XOC logs. The current implementation supports everything presented in this paper. All OCBC tools are available as plug-ins for the ProM framework and can be obtained by downloading ProM from `promtools.org` and installing the *OCBC package* using ProM's package manager.[11] In the remainder, we describe the functionality of the currently available plug-ins.

The *OCBC Model Editor* shown in Figure 18 can be used to create, modify, and view OCBC models. The editor supports all elements mentioned in Definition 11 (class model, activities, behavioral constraints, activity-class relations, cardinality constraints, etc.). Figure 18 shows the model used in the introduction (cf. Figure 2). Models can be imported and exported.

The *XOC Log Inspector* shown in Figure 19 is used to visualize *XOC event logs*. The XOC logs are, just like OCBC models, a new type of object in ProM. The XOC logging format implements the event log notion defined in Definition 10. XOC logs are stored in a new XML format.

[10] That is, the complexity measured in the size of the log only, assuming that the OCBC model is fixed.

[11] Access *http://www.win.tue.nl/ocbc/* for more information, such as tools, manuals, OCBC models and XOC logs mentioned in this paper.

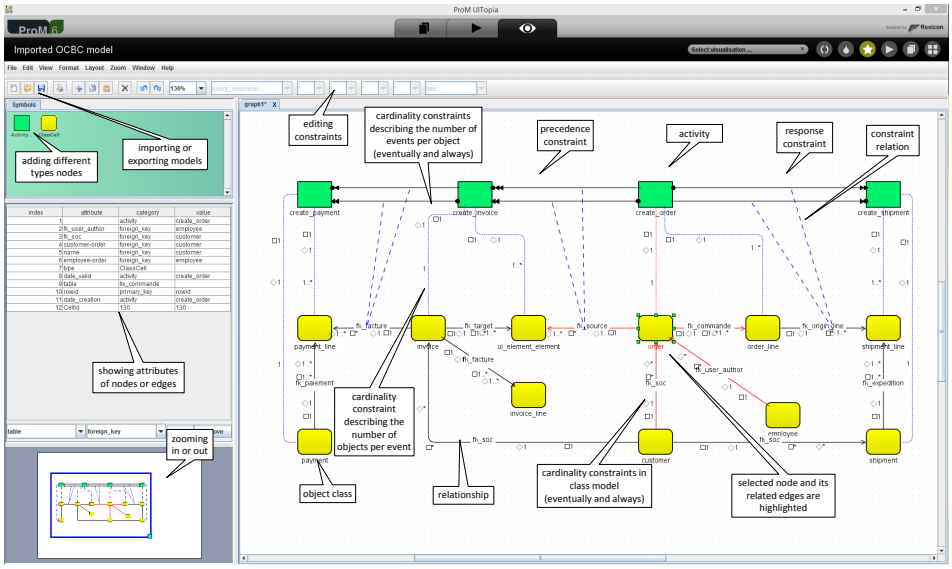

Fig. 18. The OCBC model editor and viewer in ProM.

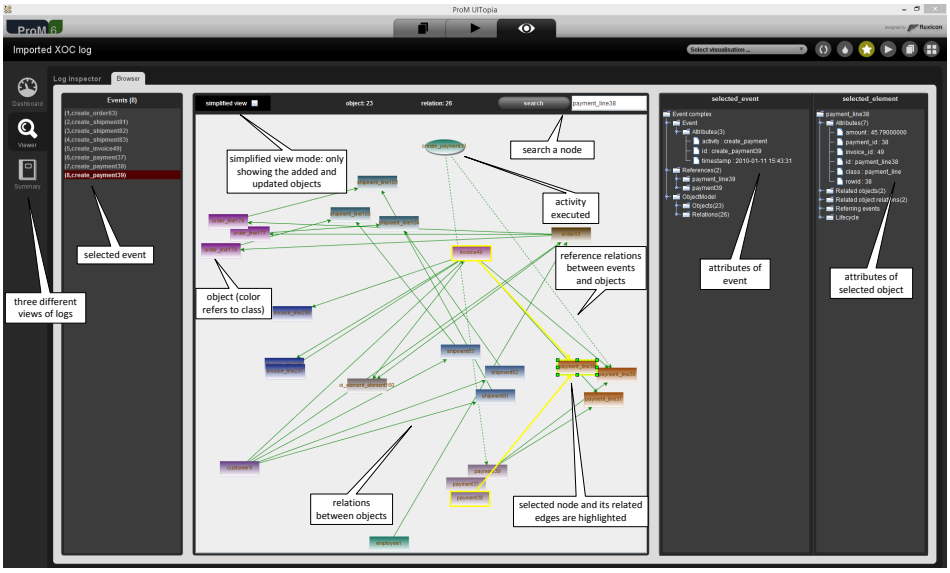

Fig. 19. ProM Plug-in to inspect XOC logs.

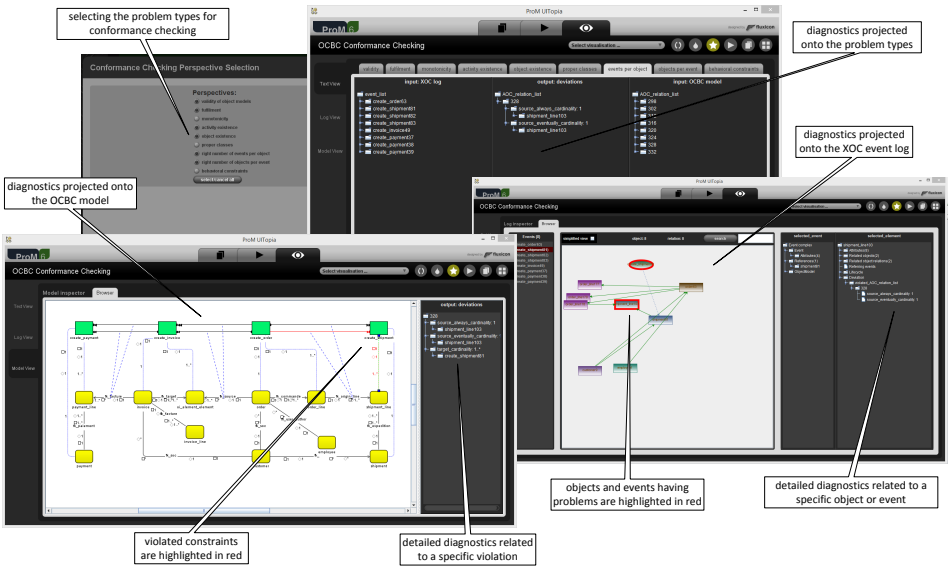

Fig. 20. The OCBC Conformance Checking plug-in provides an interface to customize problem types for conformance checking and three conformance views (type view, log view, and model view) to display the diagnosis.

The *OCBC Conformance Checking* plug-in implements the different checks described in Section 6. The plug-in takes an OCBC model and a XOC event log as input. The user can select the perspectives to be checked when starting the plug-in. (This step has been added because some checks are more time-consuming than others.) By selecting all perspectives, each of the checks described in Definition 12 is performed. After all checks have been performed, three views are provided:

1. **Diagnostics projected onto the problem types**: A high-level summary of the conformance problems are found classified in problem types (i.e., Type I-IX in Definition 12). The display of each problem type (e.g., right number of events per object) consists of three parts, i.e., the input from the XOC log (e.g., events and corresponding object references) on the left, the input from the OCBC model (e.g., AOC relations) and the output of the diagnosis result (e.g., violated AOC relations with deviating objects) in the middle.
2. **Diagnostics projected onto the XOC event log**: The OCBC Conformance Checking plug-in provides a view similar to the XOC Log Inspector (Figure 19), but now highlighting problematic events and objects. The events and objects that have problems get a red border and clicking on these provides a explanation of the problem (e.g., the constraint violated).
3. **Diagnostics projected onto the OCBC model**: Problems can also be mapped onto the OCBC model. A view similar to the OCBC Model Editor (Figure 18) We highlight the parts of the model where deviations occurred. Clicking on the problem

provides detailed diagnostics. For example, by clicking on a violated behavioral constraint the user can see a list of all reference events for which the constraint was violated.

Figure 20 shows the three views provided by the OCBC Conformance Checking plug-in. It illustrates that conformance problems can be viewed from different angles. Figure 21 gives more details about how each view describes the 9 types of diagnosis results. Note that the type view displays the diagnosis result in a hierarchy structure and each number in the square annotations, e.g., "(1)", indicates the hierarchy level of the elements (e.g., events and objects) after the number. For instance, the annotation on the top left, i.e., the description about how the type view displays the diagnosis result of "object existence", specifies a hierarchy, in which the first level shows deviating events and the second level shows inexistent objects for each event.

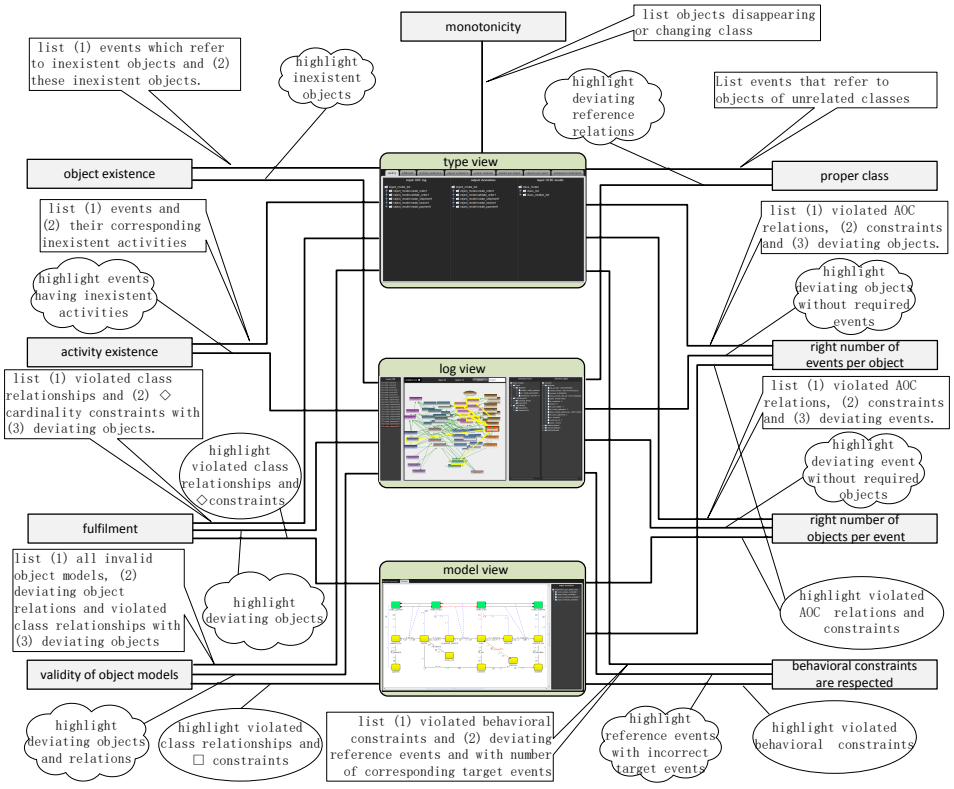

Fig. 21. The description about how the three conformance views display the 9 types of diagnosis problems (the square notation corresponding to the type view, the cloud notation corresponding to the log view, and the ellipse notation corresponding to the model view).

Next to the plug-ins mentioned there is also a plug-in extracting XOC event logs from database tables and a plug-in discovering OCBC models from XOC event logs (refer to our paper "Automatic Discovery of Object-Centric Behavioral Constraint Models"). We anticipate to develop more plug-ins supporting OCBC models and/or XOC logs. However, this is outside the scope of this paper where we focus on OCBC conformance checking.

8 Case Study Using Dolibarr ERP/CRM

The conformance checking approach highly depends on the availability of XOC event logs following the event log notion defined in Definition 10. Such event logs are different from standard XES, MXML, and CSV log files in two respects: (1) there is no single case notion and (2) each event is related to an object model describing the "state" of the process. This aligns well with the way that actual information systems work: data is stored in a database and transaction update the database. As mentioned in the introduction, such information can be obtained from enterprise systems provided by vendors such as SAP (S/4HANA), Microsoft (Dynamics 365), Oracle (E-Business Suite), and Salesforce (CRM).

Here we use *Dolibarr ERP/CRM* to illustrate the feasibility of the approach and the availability of the data assumed. Dolibarr ERP/CRM is an open source software package for small and medium companies (`www.dolibarr.org`). It includes all the main features of an ERP/CRM suite, except for advanced accountancy functions. For example, Dolibarr ERP/CRM supports sales, orders, procurement, shipping, payments, contracts, project management, etc. We use Dolibarr ERP/CRM because it provides an API that makes it easy to perform controlled experiments. Other open-source Enterprise Resource Planning (ERP) systems include Odoo, ERPNext, iDempiere, webERP, Openbravo, and Opentaps. Dolibarr ERP/CRM is often named as one of the leading open-source ERP systems and has been downloaded over a million times. It is mostly used by smaller organizations, foundations, and freelancers.

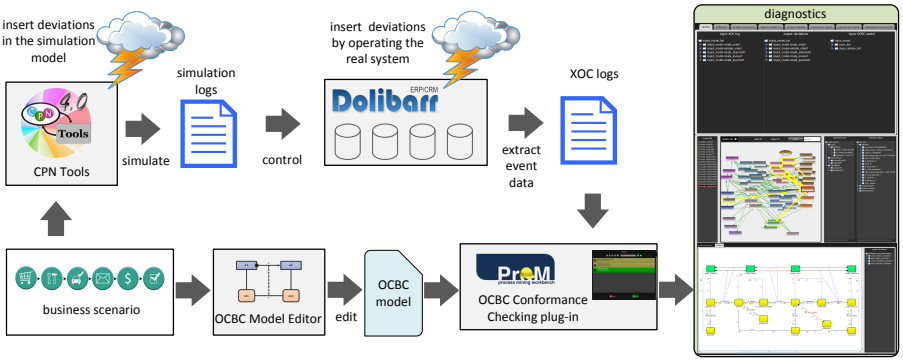

Fig. 22. The approach to evaluate the OCBC conformance checking approach and tooling.

In order to conduct controlled experiments. We used the approach shown in Figure 22. Given a particular process scenario, we create both a normative OCBC model with *OCBC Model Editor* and a simulation model with CPN Tools (`cpntools.org`). (See [**?**,**?**] for an introduction to modeling using Colored Petri Nets (CPNs) and the CPN Tools environment.) Through simulating complex process involving multiple interacting entities, we can control the number and type of deviations from the normative OCBC model, resulting in a simulation log. We interpret the simulation log to automatically operate the Dolibarr ERP/CRM and populate the corresponding database.[12] For example, if an order is created in the simulation and exported in the simulation log, it is also created in the real system. By running the simulation, the tables of Dolibarr get filled with information about orders, customers, deliveries, etc. After running Dolibarr for some time we extract XOC event logs from the database of Dolibarr. The XOC event logs extracted from Dolibarr and the OCBC model are then loaded into ProM for conformance checking.

In this paper we do not elaborate on the extraction of XOC event logs from Dolibarr. Each process requires the selection of the relevant tables. There is no fully automated way to do this. Knowledge of the ERP system and the process being analyzed are needed, e.g., the expertise of the system can help us identify activities and the *redo logs* can help us reconstruct object models (i.e., historic states of process). This is consistent with current practice in industry. There are several process mining tool vendors and consultancy firms targeting SAP users. The extraction process is highly repeatable for the more common SAP processes.[13]

The approach depicted Figure 22 allows us to check whether the deviations that are injected can actually be discovered. To illustrate the approach, let us focus on the *order-to-cash* process in Dolibarr. We take Figure 23 as the normative model of the process. The model indicates that there are nine classes and four activities involved in this process. The ten class relationships (i.e., $r1 \sim r10$) reveal the constraints between classes, e.g., each order line should eventually have a corresponding shipment line indicated by $r9$ (this is consistent with the real scenario where each order line is eventually shipped to the corresponding customer). The seven behavioral constraints (i.e., $c1 \sim c7$) present restrictions assigned on the temporal order between events of different activities. For instance, $c6$ indicates that each "create order" event is followed by one or more corresponding "create shipment" events while $c7$ requires each "create shipment" event is preceded by precisely one corresponding "create order" event since Dolibarr does not enable creating shipments covering multiple orders. The eight AOC relations (i.e., ① $\sim$ ⑧) specifies the cardinaltiy constraints between activities and classes. For example, ⑤ shows a one-to-one correspondence between "create order" events and "order" objects, i.e., if an "order" object is observed, the corresponding "create order" activity needs to be executed once and vice versa.

Based on the normative model, we can add deviating behavior into normal behavior and check whether this is picked up. In our approach, deviations can be added by two methods:

[12] More information is at http://www.win.tue.nl/ocbc/softwares/data_generation.html.
[13] An approach for extraction is at http://www.win.tue.nl/ocbc/softwares/log_generation.html.

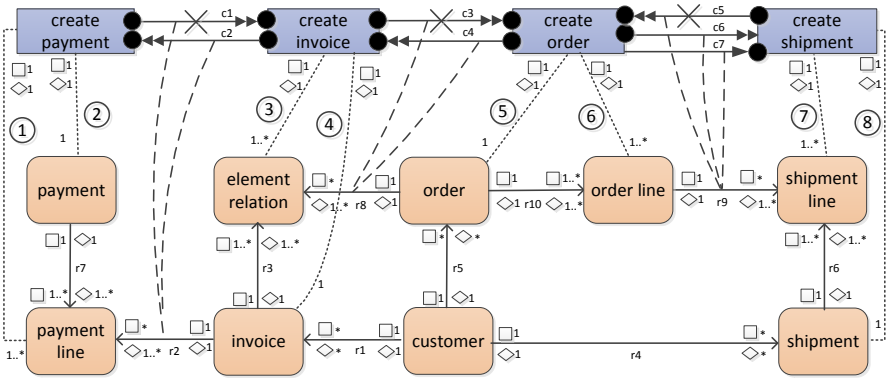

Fig. 23. The normative OCBC model of the *order-to-cash* process in Dolibarr.

– *Adding deviation paths into the simulation model.* This method is used to add deviations on the model level. More precisely, we can model the deviating behavior as a path violating the normative model. For instance, we can add a alternative path which can skip the activity "create shipment" when some attributes of orders satisfy predefined rules. This method generates normal behavior and deviating behavior at the same time when the model is simulated, which enable generating large numbers of deviations.
– *Adding deviating behavior manually on the real system.* This method is used to add deviations on the instance level, i.e., operating the real system deliberately violating the the normative scenario. For instance, after creating an order, we never create shipments for this order. This method make it possible to intertwine the simulated data with real behavior, i.e., we create the normal behavior through simulate the model and insert deviations on the real system.

In our experiments, we employ the second method to add typical deviations which may really happen in daily transactions. Since OCBC models cover the the behavioral perspective, the data perspective and the interactions between two perspectives, we create deviations of three categories accordingly.

– *Deviations related to the behavioral perspective.* Like other modeling language such as Petri nets, OCBC models supports checking conformance on the behavioral perspective. In the normal scenario, each "create order" event is followed by at least one "create invoice" event, indicated by the constraint $c4$. In the experiment, we add a deviation violating $c4$, i.e., an "create order" event is never followed by corresponding "create invoice" events.
– *Deviations related to the data perspective.* A challenge for detecting behavioral deviations is that how to detect implicit deviations. For instance, an "create order" event has corresponding "create shipment" events but does not have sufficient ones, i.e., order lines created by the "create order" event do not totally be shipped to customer. This implicit deviation is indeed violating our scenario but satisfy the behavioral constraint (i.e., $c6$, which requires a one-to-many relation between "cre-

ate order" events and "create shipment" events). Since OCBC models have a data perspective, it is possible to transform such implicit deviations onto the data perspective. For example, the deviation mentioned above can be interpreted as some order lines have no corresponding shipment lines, and be detected through checking the cardinality constraints on the class relationship $r9$.

– *Deviations related to the interactions between two perspectives.* In the Dolibarr system, when an invoice is created, it is normally linked to one or more existing orders. As a result, one or more element relations (showing the correspondence between the invoice and the orders) are created when an "create invoice" event happens, which is indicated by the cardinality "1..*" of the AOC relation ③. In reality, a common deviating situation is that one forgets to link one invoice to any orders. In our experiment, we mimic this situation, i.e., create an invoice without any element relations, resulting in a deviation violating the constraints (i.e., "1..*" of ③) on the interactions of two perspectives.

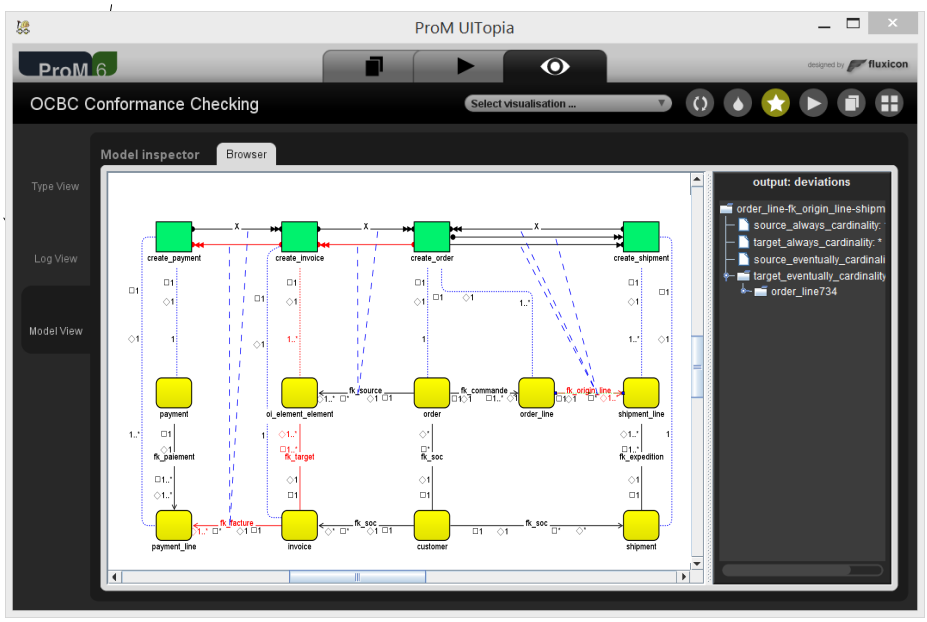

Fig. 24. Needs to be a replaced by a more realistic scenario.

Taking an XOC log with the deviations illustrated above and the normative OCBC model as input, we use the "OCBC Conformance Checking" plug-in to check the conformance between the log and the model and the result is shown in Figure.

9 Conclusion

In this paper, we proposed *Object-Centric Behavioral Constraint* (OCBC) models as a means to graphically model control-flow and data/objects in an integrated manner. Cardinality constraints are used to *specify structure and behavior in a single diagram*. In existing approaches, there is often a complete separation between data/structure (e.g., a class model) and behavior (e.g., BPMN, EPCs, or Petri nets). In OCBC models, different types of instances can interact in a fine-grained manner and the constraints in the class model guide behavior.

OCBC models are particularly suitable for conformance checking. Many deviations can only be detected by considering multiple instances and constraints in the class model. In this paper, we identified nine types of conformance problems that can be detected using OCBC models.

This paper also presented three ProM plug-ins supporting OCBC models and the corresponding XOC event logs. These serve as a proof-of-concept. All types of deviations described in this paper can be detected automatically using our software. Moreover, the paper illustrates that the data needed for such checks are available in today's information systems. Several experiments were conducted using Dolibarr. This open-source ERP/CRM system has the type of information typically found in other ERP/CRM systems. In order to perform controlled experiments, CPN Tools was used to control Dolibarr via an API. By using simulation we could play different scenarios on the actual ERP/CRM and inject deviations. The OCBC Conformance Checker plug-in was able to detect all such deviations and provide adequate diagnostics.

This paper serves as a starting point for a new line of research. In fact, there are many possible avenues of *future work*.

We already started working on the *discovery* of OCBC models from event logs. The initial results look very promising.

We would also like to improve the *performance* of the current conformance checking tools. Obviously, it is sufficient to just store the object references and updates in the event log. This will make the approach much better scalable. Because constraints can be checked separately, there are different approaches possible to further improve performance. However, the focus of the current was was on functionality rather than performance, i.e., answering questions that cannot be answered using existing approaches.

We also want to identify typical behavioral (anti-)*patterns* that involve multiple instances or interaction between structure and behavior. Figure 25 shows an example pattern. Along this line, we plan to study the effect of introducing *subtyping* in the data model, a constraint present in all data modeling approaches. The interplay between behavioral constraints and subtyping gives rise to other interesting behavioral patterns. For example, *implicit choices* may be introduced through subtyping. Consider a response constraint pointing to a *payment* class with two subclasses *credit card payment* and *cash payment*. Whenever the response constraint is activated and a payment is expected, such an obligation can be fulfilled by either paying via cash or credit card.

Finally, we also want to investigate how the notions of *consistency* and *constraint conflict/redundancy*, well-known in the context of Declare [**?**], and the corresponding

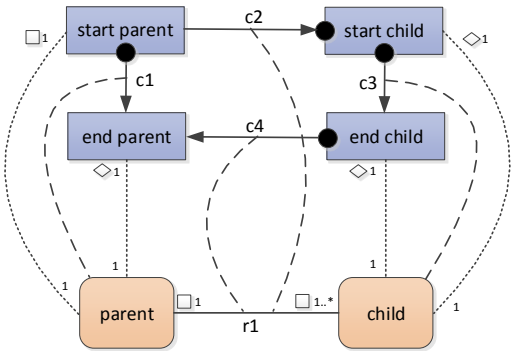

Fig. 25. Example pattern. After starting the parent, all k children (as defined by $r1$) need to start. After all k children ended, the parent ends.

notions of *consistency* and *class consistency*, well-known in data models [**?**], can be suitably reconstructed and combined in our setting.